UNIVERSITÉ DE MONTRÉAL

ACRE : UN GÉNÉRATEUR AUTOMATIQUE D'ASPECT POUR TESTER DES LOGICIELS ÉCRITS EN C++

ETIENNE DUCLOS
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2012

© Etienne Duclos, 2012.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ACRE : UN GÉNÉRATEUR AUTOMATIQUE D'ASPECT POUR TESTER DES LOGICIELS ÉCRITS EN C++

présenté par : DUCLOS, Etienne.
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. ADAMS Bram, Ph.D., président.
M. GUÉHÉNEUC Yann-Gaël, Doct., membre et directeur de recherche.
M. LE DIGABEL Sébastien, Ph.D, membre et co-directeur de recherche.
M. ORBAN Dominique, Doct., Sc., membre.

À mon compte en banque dégarni,
sans qui je serais déjà en tour du monde . . .

REMERCIEMENTS

Je tiens tout d'abord à remercier toutes les personnes qui m'ont permis de venir à l'École Polytechnique de Montréal pour réaliser ma maîtrise. Je remercie ainsi chaleureusement Sébastien Le Digabel et Yann-Gaël Guéhéneuc, mes directeurs de recherche, qui m'ont appuyé et soutenu dans mes démarches, notamment administratives, et qui ont tout fait pour que ma maîtrise et mon séjour au Québec se passent dans les meilleurs conditions possibles. Je remercie aussi Christophe Duhamel, directeur de filière à l'ISIMA, qui m'a lui aussi appuyé lors de mes démarches.

Je tiens à remercier aussi tous les membres des laboratoires ptidej et soccerlab pour leur accueil et leur bonne humeur, qui m'ont permis de travailler dans une bonne ambiance et de découvrir de nouvelles cultures. Je remercie notamment Neelesh Bhattacharya, avec qui j'ai collaboré sur un projet de recherche. Je tiens également à remercier les professeurs Giuliano Antoniol, Giovanni Beltrame et Bram Adams pour leurs conseils sur mes différents projets.

Je remercie finalement tous mes amis et ma famille, qui m'ont aidé à décompresser et à me changer les idées lorsque j'en avais besoin, et ainsi de finir ma maîtrise encore sain d'esprit, si tant est que je l'étais au moment de la commencer.

RÉSUMÉ

La qualité d'un logiciel est une notion très importante de nos jours. Un des moyens de s'assurer de la bonne qualité d'un logiciel est de le tester afin d'en détecter, et de corriger, le plus grand nombre de bogues possible.

L'approche proposée dans ce mémoire de maîtrise consiste à utiliser la programmation orientée aspect (AOP, pour *Aspect Oriented Programming*) afin de tester des programmes écrit en C++. Elle consiste de plus en la création d'un langage dédié (DSL, pour *Domain Specific Language*) et d'un logiciel capable de le lire, ACRE (**A**utomatic aspe**C**t c**RE**ator). Ces deux outils permettent la génération automatique d'aspects écrits en AspectC++, aspects permettant d'effectuer des tests de mémoire, d'invariants ou d'interférence dans des programmes écrits en C++.

La programmation orientée aspect est un paradigme basé sur le principe de la séparation des préoccupations, qui veut que chaque module d'un programme s'occupe uniquement d'une préoccupation et ne s'implique pas dans les autres. L'AOP a déjà été utilisée afin de tester des logiciels, mais uniquement pour des programmes écrits en Java et pour des programmes embarqués écrits en C++. De plus très peu de solutions ont été proposées afin d'automatiser la création des aspects et leur utilisation. Lorsque de telles solutions ont été proposées, elles l'étaient uniquement pour des programmes écrits en Java. Notre approche permet donc la génération automatique d'aspects pour des programmes écrits en C++. Ceci est fait grâce au DSL non-intrusif (dans le sens qu'il ne modifie pas le comportement du code source) introduit sous forme de commentaires dans le code source du programme à tester. L'écriture des lignes de DSL au plus près du code à tester est par ailleurs la seule modification requise à effectuer dans le code source du programme. Les aspects générés sont écrits dans des fichiers séparés et sont ensuite tissés au moment de la compilation avec le code source du programme. Notre approche permet ainsi aux développeurs de pouvoir tester un programme sans avoir à en modifier le comportement.

Une première étude de cas pour notre approche est son application afin de tester NOMAD, logiciel d'optimisation de fonctions de type boîte noire, utilisé tant en recherche qu'en industrie. Nous avons donc généré automatiquement des aspects permettant la découverte de bogues dans NOMAD. Un test de mémoire nous a permis de détecter une fuite mémoire importante, rapportée par un relevé de bogues d'un utilisateur, grâce à un aspect généré automatiquement.

Une deuxième étude de cas nous a permis de rassurer l'équipe de développement de NOMAD. Nous avons en effet pu vérifier la validité d'un point mathématique crucial de

l'algorithme implémenté. Ce test d'invariant a été là encore effectué grâce à un aspect généré automatiquement et sans modifier le comportement du code source de NOMAD.

Une troisième étude de cas concerne la détection de bogues de type interférence dans des programmes parallèles. Un bogue de type interférence se produit lorsque qu'au moins deux fils d'exécution d'un programme tentent d'écrire en même temps dans une même variable. Ce type de bogue fait partie des bogues affectant les programmes parallèles les plus difficiles à détecter et peu de logiciels actuels permettent de les découvrir à coup sûr. Notre approche permet l'obtention, grâce à des aspects, des temps d'accès en lecture et en écriture d'attributs du programme, ces temps étant ensuite utilisés par une approche mathématique qui aide les logiciels actuels dans leur détection de ce type de bogue.

Il y a cependant des limites à l'utilisation de notre approche, notamment dûes à des limitations inhérentes à AspectC++, limitations dont nous discutons.

ABSTRACT

In the modern era, software quality is very important. In order to have high software quality, we can test it to find and remove the maximum number of bugs.

Our approach uses Aspect-Oriented Programming (AOP) to test software written in C++. We propose a Domain Specific Language (DSL) and ACRE, an **A**utomatic aspe**C**t c**RE**ator that can read our DSL. From these two tools we can automatically generate aspects written in AspectC++ to perform memory, invariant and interference testing for software written in C++.

Aspect-Oriented Programming is a paradigm based on the concept of separation of concerns. In this concept, each module deals with one and only one concern, and does not deal with the concerns of other modules. AOP has already been used in testing, but only for software written in Java or embedded software written in C++. Few of these approaches allow an automatic generation of aspects, and to date, it was only to test software written in Java. Our approach allows automatic aspect generation for software written in C++ thanks to our non-intrusive DSL (in the sense that it does not change the behavior or the source code). Our DSL is inserted in the source code of the tested software as comments and this is the only modification made in the source code. Generated aspects are then written to dedicated files and woven within the source code at compile time. Our approach allows developers to test software without having to modify its behavior.

Our first case study is to apply our approach with the intent of testing NOMAD, a black box optimization software used both in research and industry. Using automatically generated aspects, we are able to do memory testing in NOMAD and locate an important memory leak mentioned in a bug report.

Our second case study is to do invariant testing in NOMAD. We are able, still using a generated aspect, to verify a crucial mathematical point of the implemented algorithm without modifying the behavior of the source code of NOMAD.

The third case study concerns interference testing in multi-threaded programs. An interference bug occurs when at least two threads try to write to the same shared variable at the same time. This kind of bug is one of the most difficult to detect and no software is able to them with 100% accuracy. Our approach is used to get both read and write access times for the attribute of the tested software using aspects. These times are then used in a mathematical approach that helps the actual software find interference bug patterns.

However there are some limitations to our approach, due to inherent limitations of AspectC++. These limitations are discussed in this thesis.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX x

LISTE DES FIGURES xi

LISTE DES EXTRAITS DE CODE xii

LISTE DES SIGLES ET ABRÉVIATIONS xiii

Chapitre 1 INTRODUCTION 1

Chapitre 2 REVUE DE LITTÉRATURE 4
- 2.1 UTILISATION DE L'AOP EN TEST 4
- 2.2 UTILISATION DE L'AOP POUR TESTER DES PROGRAMMES ÉCRITS EN JAVA 5
- 2.3 UTILISATION DE L'AOP POUR TESTER DES PROGRAMMES EMBARQUÉS ÉCRITS EN C++ 6
- 2.4 PROBLÈMES LIÉS À L'UTILISATION DE L'AOP 7
- 2.5 TEST D'INVARIANT 7
- 2.6 TESTS D'INTERFÉRENCE 8

Chapitre 3 CONTEXTE 9
- 3.1 PROGRAMMATION ORIENTÉE ASPECT 9
- 3.2 GESTION DE LA MÉMOIRE EN C++ 11
- 3.3 TESTS D'INVARIANT 12
- 3.4 BOGUES DE TYPE INTERFÉRENCE 13

Chapitre 4 APPROCHE 15
4.1 DSL 15
4.1.1 *COUNTER* 16
4.1.2 *LOGGING* 17
4.1.3 *TIMING* 18
4.1.4 *CHECKING* 19
4.2 ACRE 22

Chapitre 5 ÉTUDE EMPIRIQUE 24
5.1 HYPOTHÈSES DE RECHERCHE 24
5.2 NOMAD 25
5.3 VALIDATION DE L'HYPOTHÈSE 1 31
5.4 VALIDATION DE L'HYPOTHÈSE 2 32
5.5 VALIDATION DE L'HYPOTHÈSE 3 35
5.6 DISCUSSION DES RÉSULTATS 38

Chapitre 6 CONCLUSION 40
6.1 SYNTHÈSE DES TRAVAUX 40
6.2 LIMITATIONS DE LA SOLUTION PROPOSÉE 41
6.3 AMÉLIORATIONS FUTURES 42

RÉFÉRENCES 43

LISTE DES TABLEAUX

Tableau 2.1 Comparaison des différents travaux existants. 7
Tableau 4.1 Synthèse du DSL. 21
Tableau 5.1 Algorithme MADS simplifié, inspiré de Le Digabel (2011). 28
Tableau 5.2 Exemple de paramètres d'entrée pour NOMAD, inspiré de Le Digabel et Tribes (2011). 29
Tableau 5.3 Exemple de sortie générée par NOMAD, inspiré de Le Digabel et Tribes (2011). 30

LISTE DES FIGURES

Figure 3.1 Exemple de comportement normal. 14
Figure 3.2 Exemple de comportement anormal. 14
Figure 5.1 Courbe des téléchargements de NOMAD depuis 2008, selon 3 sources. 26
Figure 5.2 Exemple de différentes configuration de treillis, tiré de Le Digabel (2011) (Reproduction avec autorisation). 27
Figure 5.3 Sortie de NOMAD avec un aspect de type *counter* tissé. 33

LISTE DES EXTRAITS DE CODE

Extrait de code 3.1 Exemple d'aspect permettant le traçage 10
Extrait de code 3.2 Code source du programme jouet 12
Extrait de code 4.1 Code utilisé pour générer un aspect de type *counter* 16
Extrait de code 4.2 Code utilisé pour générer un aspect de type *logging* 17
Extrait de code 4.3 Code utilisé pour générer un aspect de type *timing* 19
Extrait de code 4.4 Code utilisé pour générer un aspect de type *checking* . . . 20
Extrait de code 5.1 Aspect permettant de trouver la fuite mémoire dans le programme jouet section 3.2 32
Extrait de code 5.2 Lignes permettant l'initialisation de la variable de l'aspect MyToyCounter 32
Extrait de code 5.3 DSL requis pour créer automatiquement un aspect trouvant la fuite dans le programme jouet 33
Extrait de code 5.4 Aspect généré automatiquement permettant la découverte d'une fuite mémoire dans NOMAD 34
Extrait de code 5.5 Lignes de code DSL requise pour générer l'aspect 5.6 . . . 36
Extrait de code 5.6 Aspect de type *timing* généré automatiquement 37
Extrait de code 5.7 Aspect de type *checking* généré automatiquement 39

LISTE DES SIGLES ET ABRÉVIATIONS

ACRE	**A**utomatic aspe**C**t c**RE**ator
AOP	Aspect-Oriented Programming
API	Application Programming Interface
DSL	Domain Specific Language
NOMAD	Nonlinear Optimizer with the Mesh Adaptive Direct search algorithm
OOP	Object-Oriented Programming

Chapitre 1

INTRODUCTION

Dans une société de plus en plus informatisée, la qualité et la fiabilité des programmes informatiques sont deux facteurs de leur qualité extrêmement importants. La maintenance de ces programmes, qui comprend entre autre la correction des différents bogues, représente quant à elle une part non négligeable du coût de développement d'un logiciel, comme expliqué par Lienz et Swanson (1980). Un moyen efficace de réduire ces coûts et de diminuer les risques de défaillance des programmes est de les tester et d'en trouver les bogues le plus tôt possible. Plus un bogue est trouvé tôt lors du développement d'un logiciel, moins il est cher à réparer (Grady (1999)).

La gestion explicite de la mémoire, dûe entre autre à l'utilisation des pointeurs et à l'absence de ramasse-miettes, oblige les testeurs de programmes écrits en C++ à se pencher sur des problèmes qu'on ne rencontre pas lors de tests de programmes écrits en Java. Des techniques de test ont ainsi été développées afin de détecter les problèmes engendrés par les pointeurs nuls, par les dépassements de mémoire tampon, mais aussi par les fuites de mémoires (Heine et Lam (2003), Schildt (2004) et Xu *et al.* (2008)). La plupart de ces techniques ont cependant l'inconvénient d'introduire des changements dans le code source du programme à tester. Cet inconvénient est aussi valable pour les tests d'invariant et d'interférence (Zeng *et al.* (2009), Musuvathi *et al.* (2007))

Cet inconvénient peut néanmoins être contourné par l'utilisation de paradigmes tels que la programmation orientée aspect (appelée par la suite AOP, pour *Aspect Oriented Programming*). Ce récent paradigme est basé sur le concept de séparation des préoccupations, qui veut que chaque module d'un programme, alors représenté sous forme d'aspects, n'ait qu'une seule préoccupation, et n'interfère pas avec les préoccupations des autres modules du programme. Les aspects sont ensuite tissés (intégrés) dans le code source du programme, écrit en orienté objet, lors de la compilation ou directement à l'exécution. Les créateurs de l'AOP, Kiczales *et al.* (1997), reconnaissent que celle-ci peut être utilisée pour tester des logiciels. Des recherches ont ainsi été menées sur ce sujet. Elles ne concernent cependant pour l'instant que les programmes écrits en Java et des programmes embarqués écrits en C++, et très peu proposent une génération automatique des aspects.

Nous continuons les recherches effectuées précédemment en proposant une approche permettant la génération automatique d'aspects afin de tester des programmes écrits en C++. Les aspects ainsi générés permettent de réaliser des tests de mémoire, qui permettent la

détection de fuites mémoire ou de problèmes d'accès à la mémoire, des tests d'invariants, qui permettent de s'assurer que les variables testées ont la bonne valeur au bon moment, et enfin des tests d'interférence, qui permettent de vérifier qu'aucune action n'interfère avec une autre. Cette approche a été soumise à la 19e conférence sur la rétro-ingénierie (Duclos *et al.* (2012), WCRE [1]).

Afin de générer automatiquement les aspects, nous proposons un langage dédié (appelé par la suite DSL, pour *Domain Specific Language*), se présentant sous forme de commentaires à introduire dans le code source et permettant la description des aspects directement dans le code source du programme à tester. Nous introduisons aussi un programme, ACRE, pour **A**utomatic aspe**C**t c**RE**ator, capable de lire ce langage et d'en générer automatiquement les aspects correspondants. Les aspects ainsi générés permettent aux programmeurs de tester leurs programmes sans modifier le comportement de leur code source, les aspects étant tissés dans le programme lors de la compilation de ce dernier. De plus, malgré la présence des lignes de DSL dans le code source du programme, ce dernier est toujours compilable comme avant.

Dans ce mémoire sont proposées les trois hypothèses suivantes :

- **Hypothèse 1 :** *L'AOP peut être utilisée efficacement pour tester des programmes C++.*
- **Hypothèse 2 :** *Des aspects peuvent être générés automatiquement depuis le code source de programmes C++.*
- **Hypothèse 3 :** *Les aspects ainsi générés peuvent être utilisés pour aider à la réalisation de tests d'interférence.*

Afin de valider ces hypothèses, nous verrons dans ce mémoire trois études de cas sur lesquels nous avons appliqué notre approche. Tout d'abord l'utilisation d'aspects automatiquement générés pour des tests de mémoire menant à la détection d'une fuite mémoire dans un programme réel, utilisé aussi bien en recherche qu'en industrie, NOMAD. Un test d'invariants, permettant la vérification de la validité d'un point mathématique crucial de l'algorithme de NOMAD constitue notre deuxième cas d'étude. Le troisième cas d'étude est quant à lui la détection de bogues de type interférence. Ces bogues surviennent dans des programmes parallèles lorsqu'au moins deux fils d'exécution tentent d'écrire en même temps dans une même variable. L'utilisation d'aspects permet l'obtention des temps d'accès en écriture et en lecture de variables. Ces temps sont ensuite utilisés dans une méthode, développée lors de ma maîtrise (Bhattacharya *et al.* (2012)), et qui permet la localisation de la position optimale pour injecter un délai dans le flux d'exécution du programme, ainsi que la durée optimale de ce délai, permettant la maximisation des possibilités d'obtention de bogues de type interférence.

Ce mémoire est organisé selon le plan suivant : le chapitre 2 présente une revue de

1. *Working Conference on Reverse Engineering*

littérature traitant tout d'abord de l'utilisation de la programmation orientée aspect dans le domaine du test de logiciel au sens large, en section 2.1, puis pour tester plus spécifiquement des programmes écrits en Java, en section 2.2, et des programmes embarqués écrits en C++, en section 2.3. La section 2.4 traite quant à elle des problèmes engendrés par l'utilisation de l'AOP. Une synthèse des travaux concernant les tests d'invariant et d'interférence, en sections 2.5 et 2.6, conclut la revue de littérature. Le chapitre 3 met en avant différentes notions nécessaires pour la compréhension de l'approche. Tout d'abord la section 3.1 introduit la programmation orientée aspect. La section 3.2 traite des problèmes liés à la gestion de la mémoire en C++, tandis que la section 3.3 explique ce que sont les tests d'invariants. La section 3.4 termine le chapitre en introduisant les tests d'interférence. Le chapite 4, via les sections 4.1 et 4.2, décrit la solution proposée, en présentant respectivement le DSL créé et le logiciel, ACRE, permettant de le lire et de générer automatiquement les aspects qui y sont décrits. Le chapitre 5 présente tout d'abord en section 5.1 les hypothèses que nous avons formulées. La section 5.2 parle de NOMAD, le logiciel auquel nous avons appliqué notre approche. Les résultats obtenus sont exposés dans les sections 5.3, 5.4 et 5.5, vérifiant chacune une hypothèse. La section 5.6 discute ces résultats. Finalement le chapitre 6 conclut le mémoire, le synthétisant tout d'abord en section 6.1, avant de parler des limitations de l'approche proposée en section 6.2 et des améliorations futures à apporter en section 6.3.

Chapitre 2

REVUE DE LITTÉRATURE

Ce chapitre résume les principaux travaux concernant l'utilisation de la programmation orientée aspect pour le test logiciel. La section 2.1 se concentre tout d'abord sur un cadre général, synthétisant les avantages et les inconvénients de l'utilisation d'aspects pour le test. Les sections 2.2 et 2.3 traitent ensuite plus particulièrement des deux directions les plus explorées dans la littérature : le test de programmes écrits en Java et le test de programmes embarqués écrits en C++. Le tableau 2.1 synthétise et compare ces différents travaux, incluant notre approche. La section 2.4 soulève les problèmes que peut apporter l'utilisation de l'AOP. Finalement les sections 2.5 et 2.6 exposent les techniques existant actuellement pour effectuer des tests d'invariant et détecter des bogues de type interférences.

2.1 UTILISATION DE L'AOP EN TEST

Lorsque la programmation orientée aspect est utilisée pour tester des logiciels, elle l'est entre autres dans le cas de tests unitaires et de tests d'intégration. On appelle test unitaire les techniques permettant de s'assurer du bon fonctionnement de parties précises d'un programme, telles les méthodes ou les classes. Ainsi, dans le cas d'une méthode, un test unitaire permet de vérifier que, pour une entrée donnée, la méthode fournit la sortie désirée. Les éléments de remplacement, ou *stubs*, sont utilisés pour simuler les composant dont la méthode testée dépend et qui ne sont soit pas encore développés, soit trop lourds pour être utilisés en test. Les pilotes, ou *drivers*, ont le même rôle que les éléments de remplacement, mais pour des éléments dépendant de la méthode testée. Les tests d'intégration font suite aux tests unitaires. Une fois que toutes les parties du module ont été testées indépendamment, elles sont intégrées et testées ensemble, pour s'assurer du bon fonctionnement du module en lui-même.

Li et Xie (2009) ont utilisé l'AOP pour des tests unitaires et d'intégration et ont conclu que les aspects permettent de faire de bons *stubs* et *drivers*. Ces conclusions viennent en partie du fait que l'utilisation d'aspects permet de ne pas modifier le code source du programme à tester. Cet avantage a aussi été discuté par Metsä *et al.* (2008), qui ont comparé l'utilisation d'aspects dans le test de logiciels et l'utilisation de techniques plus classiques, telles les macros ou les interfaces de test. Ils ont cependant conclu que, malgré l'avantage procuré par l'absence de modifications, l'AOP ne devrait pas être utilisée seule pour les tests unitaires et les tests

d'intégration. Ils soulignent cependant que ceci pourrait évoluer grâce au développement de logiciels permettant une meilleure utilisation de l'AOP, en proposant par exemple des aspects génériques ou pré-conçus. Ce point avait par ailleurs déjà été mentionné par Metsä *et al.* (2007), qui pointaient les deux principaux freins à l'utilisation de l'AOP dans le domaine du test : le manque d'outils ou d'interfaces permettant une utilisation simple des aspects (ce à quoi notre approche répond) et le tissage des aspects qui est une étape supplémentaire dans le processus de construction du programme. À ces deux points, Farhat *et al.* (2010) ont rajouté le fait que l'AOP est une technologie plutôt récente que peu de testeurs connaissent, ce qui n'incite pas à son utilisation.

Nous pensons que notre approche permet de contourner ces désavantages. ACRE rend possible l'utilisation d'aspects sans avoir besoin de connaissances en programmation orientée aspect, et fournit une interface, grâce au DSL que nous avons développé, qui permet de générer automatiquement les aspects.

2.2 UTILISATION DE L'AOP POUR TESTER DES PROGRAMMES ÉCRITS EN JAVA

L'utilisation de la programmation orientée aspect dans le domaine du test s'est essentiellement faite pour les logiciels écrits en Java, souvent couplée avec l'utilisation de JUnit [1], une bibliothèque de tests unitaire pour le langage Java. Xu *et al.* (2004) ont créé un langage de description de tests orientés aspect et un cadre d'applications (*framework*), JAOUT, capable de le lire. Leur approche permet la génération d'aspects spécifiques au programme à tester. Ces aspects sont ensuite utilisés avec JUnit afin de tester les différentes parties du programme. Le principal avantage de JAOUT est qu'il est automatique, les aspects et les tests correspondants étant créés par leur logiciel, ce qui n'est pas le cas des autres approches.

Knauber et Schneider (2004) ont utilisé AspectJ (programmation orientée aspect pour Java) et JUnit afin de tester différentes préoccupations transverses, telles la sécurité, la gestion transactionnelle ou la gestion d'erreurs dans une ligne de production de logiciels. Ils ont retenu comme principal point positif la modularité des aspects, qui permet une meilleure réutilisation des modules de tests.

d'Amorim et Havelund (2005) ont proposé une approche utilisant entre autre l'AOP afin de vérifier le bon comportement de programmes Java lors de leur exécution. Les aspects sont utilisés pour capter les valeurs de certaines variables au fur et à mesure de l'exécution du programme. Ces valeurs sont ensuite utilisées dans des tests d'invariants afin de savoir si le programme s'exécute correctement ou non et en avertir l'utilisateur le cas échéant.

1. `http://www.junit.org/`

Cependant tous ces travaux concernent uniquement les programmes écrits en Java. Notre approche permet quant à elle d'appliquer ces techniques aux programmes écrits en C++.

2.3 UTILISATION DE L'AOP POUR TESTER DES PROGRAMMES EMBARQUÉS ÉCRITS EN C++

Pesonen *et al.* (2005), Pesonen (2006) et Metsä *et al.* (2007) ont utilisé la programmation orientée aspect pour tester des programmes embarqués, écrits en C++ et tournant sous Symbian OS. Leur choix d'utiliser l'AOP plutôt qu'une approche plus standard vient des contraintes liées à l'embarqué, telles les ressources limitées. Tout comme Li et Xie (2009), ils se sont tout d'abord concentrés sur les tests unitaires et les tests d'intégration, utilisant les aspects comme *stubs* et *drivers*.

Pesonen *et al.* (2005) ont testé des parties de leurs programmes spécifiques à l'embarqué. Ils ont ainsi utilisé l'AOP pour vérifier que les parties de leur programme liées au matériel fonctionnaient toujours de manière optimale même si certains périphériques n'étaient pas activés ou pas connectés. Leurs résultats sont cependant mitigés : des améliorations ont été montrées par rapport à des techniques plus standard, mais AspectC++, langage permettant l'utilisation de l'AOP pour le langage C++, ne leur permettait pas de faire tout ce qu'ils souhaitaient. Ils ne pouvaient ainsi pas définir des points de jonction pour toutes les parties du programmes qu'ils voulaient tester. Metsä *et al.* (2007) ont réussi à améliorer les performances de cinq types de tests fonctionnels : les tests de mémoire, de performance, de robustesse, de fiabilité et de couverture. Ils ont aussi conclu que l'utilisation d'aspects pour les tests leur a permis d'améliorer la couverture totale de leur programme, c'est-à-dire d'augmenter la quantité de lignes de code testées par rapport à l'utilisation de techniques standard.

Le tableau 2.1 synthétise les différents travaux exposés lors de cette revue de littérature. Pour chacun d'entre eux, il résume le langage dans lequel sont écrits les logiciels testés, comment s'effectue la génération des aspects (manuellement ou automatiquement), et le type de test effectué par les aspects. Nous pouvons ainsi remarquer que notre approche propose une génération automatique d'aspects permettant tout type de test pour des programmes écrits en C++, ce qui n'a pas été proposé jusqu'à présent.

Tableau 2.1 : Comparaison des différents travaux existants.

Auteurs	Langage			Génération		Types de tests	
	Java	C++ embarqué	C++	auto.	manuelle	unit. et d'int.	tout type
Xu *et al.* (2004)	X			X		X	
Knauber *et al.* (2004)	X				X	X	
Li et Xie (2009)	X				X	X	
d'Amorim *et al.* (2005)	X			X			X
Pesonen *et al.* (2005)		X			X		X
Pesonen (2006)		X			X		X
Metsä *et al.* (2007)		X			X		X
Metsä *et al.* (2008)		X			X	X	
ce travail			X	X			X

2.4 PROBLÈMES LIÉS À L'UTILISATION DE L'AOP

L'utilisation de l'AOP, et pas uniquement pour les tests, peut cependant générer quelques problèmes. Störzer *et al.* (2006) se sont penchés sur les problèmes qui surviennent lorsque plusieurs aspects sont utilisés dans un même programme et concernent la même méthode. Ils ont ainsi remarqué que lorsqu'aucun ordre n'est défini pour l'exécution des aspects, les programmes peuvent alors fournir des sorties incorrectes. Bradley (2003), qui a étudié l'utilisation de l'AOP en industrie, a entre autres déduit qu'en plus de tester le logiciel, il faut tester les aspects qu'on utilise, et que ceux-ci affectent la compréhensibilité du code, sa testabilité, mais aussi, dans certains cas, son exactitude. Cependant, ses travaux ont été effectués en 2003, et l'AOP n'offrait alors pas les possibilités actuelles pour les points de jonctions et les méthodes d'aspects. Ils pensent ainsi que dans le futur l'utilisation de l'AOP pourrait augmenter la qualité globale des logiciels.

Par la suite, Zhou *et al.* (2004) ont proposé des solutions pour tester les aspects, en plus du programme en lui même, et Xie et Zhao (2006) on créé un cadre d'application permettant de générer automatiquement les tests pour tester les aspects écrits en AspectJ.

2.5 TEST D'INVARIANT

Les tests d'invariant permettent de vérifier qu'une propriété d'un programme est toujours vraie. Ils permettent entre autre de déterminer la présence ou non de dépassements de mémoire (*buffer overflow*, Zeng *et al.* (2009)). Les invariants peuvent aussi être utilisés pour l'analyse d'un programme, comme expliqué par Nielson *et al.* (1999). Il existe différentes méthodes pour réaliser des tests d'invariant : statiques, en utilisant des patrons de concep-

tion (Gibbs *et al.* (2002)), ou dynamique, en récupérant les valeurs de certaines variables durant l'exécution du programme. Daikon (Ernst *et al.* (2006)) analyse ainsi ces variables, les comparant à des valeurs de références et indique ainsi si le programme s'exécute normalement ou non. Toutes ces approches nécessitent cependant de modifier le code source du programme pour pouvoir effectuer les tests d'invariant. Nous verrons que notre approche permet de contourner cette limitation.

2.6 TESTS D'INTERFÉRENCE

Les bogues de type interférence, qui se produisent lorsqu'au moins deux fils d'éxécution tentent d'écrire en même temps dans une même variable (voir la section 3.4 pour plus de détails), font partie des bogues les plus difficiles à détecter (Software Quality Research Group (2012)). De nombreux logiciels, tels ConTest (Edelstein *et al.* (2003)), RaceFinder (Ben-Asher *et al.* (2003)), CalFuzzer (Joshi *et al.* (2009)) ou CTrigger (Park *et al.* (2009)) ont été développés afin de détecter ce genre de bogues. Les différentes approches utilisées, telles l'énumération d'ordre d'éxécution des fils, l'énumération des états des fils ou encore des heuristiques, ne permettent cependant pas de découvrir à coup sûr tous les bogues de types interférence présents dans un logiciel. À ce jour le logiciel le plus performant pour la détection de bogues de type interférence est CHESS, développé par Musuvathi *et al.* (2007). Il contourne la plupart des limitations des précédentes approches et permet de tester des logiciels de grande taille. Cependant il n'assure pas non plus la détection de tous les bogues existants.

Nous avons développé une approche permettant l'amélioration des capacités des logiciels présentés précédemment (Bhattacharya *et al.* (2012)). Cette approche permet de découvrir l'emplacement optimal pour insérer un délai dans un programme parallèle, ainsi que la durée optimale de ce délai, afin de générer des bogues de type interférence survenant lorsque deux fils d'exécution tentent d'écrire en même temps dans une même variable.

Chapitre 3

CONTEXTE

Ce chapitre présente les notions nécessaires à la compréhension de l'approche. Tout d'abord la section 3.1 introduit la programmation orientée aspect. La section 5.2 se concentre sur NOMAD, le logiciel sur lequel les résultats ont été obtenus, alors que la section 3.2 traite des problèmes spécifiques liés à la gestion de la mémoire dans le langage C++. Finalement la section 3.4 explique ce que sont les bogues de type interférence.

3.1 PROGRAMMATION ORIENTÉE ASPECT

La programmation orientée aspect est une technique de programmation basée sur le principe de séparation des préoccupations (*separation of concerns*) et inventée par Kiczales *et al.* (1997). L'idée derrière le principe de séparation est que chaque module d'un même programme doit avoir sa propre préoccupation, telle la sécurité ou l'ouverture de sessions, et ne doit pas s'occuper des préoccupations des autres modules. Le but étant d'augmenter la modularité des programmes. La programmation orientée aspect, pour les programmes Java et C++, ne peut être utilisée seule. Une fois créés, les aspects doivent être tissés avec le code source (objet ou procédural) d'un programme pour pouvoir être utilisés.

L'exemple le plus souvent utilisé pour vanter les mérites de l'AOP est le traçage. Lorsqu'un développeur utilisant la programmation orientée objet veut que son programme affiche à chaque utilisation d'une méthode "On entre dans la méthode x", il doit modifier manuellement le corps de toutes ses méthodes afin de rajouter l'instruction d'écriture. En utilisant la programmation orientée aspect, il suffit au développeur d'écrire un aspect tel que celui dont le code est présenté dans l'extrait de code 3.1 et de tisser cet aspect dans son programme. Aucune modification n'est requise dans le code source du programme originel et le résultat final est le même : à l'exécution du programme, dès qu'une méthode est appelée, la phrase "On entre dans la méthode x", où x est la signature de la méthode, est affichée.

Les principaux langages de programmation orientée aspect sont AspectJ [1] pour Java et AspectC++ [2] pour C++. Notre approche utilisant AspectC++, c'est ce langage qui est détaillé par la suite.

Les principaux concepts d'AspectC++ sont, selon son modèle d'exécution, les *aspect*,

1. `http://www.eclipse.org/aspectj/`
2. `http://www.aspectc.org/`

Extrait de code 3.1 : Exemple d'aspect permettant le traçage.

```
#ifndef __TRACE_ASPECT_AH__
#define __TRACE_ASPECT_AH__

#include <stdio.h>
#include <stdlib.h>

aspect traceAspect{
  public:
    pointcut logMethods() = call("% %::%(...)");
    advice logMethods() : before() {
      printf("On entre dans la methode : %s\n", JoinPoint::signature());
    }
};

#endif
```

méthode d'aspect (ou *advice*), *coupe transverse* (ou *pointcut*) et *point de jonction* (ou *join point*). En AspectC++ (Spinczyk *et al.* (2005)), comme dans tous les langages orientés aspect, un aspect est l'équivalent d'une classe en programmation orientée objet. Cependant, en plus de contenir des déclarations d'attributs et de méthodes, il peut contenir des déclarations de coupes transverses et de méthodes d'aspect. Une coupe transverse, définie par le mot clé *pointcut*, est un ensemble de points de jonction correspondant à une même expression et éventuellement regroupés sous un même nom. L'expression permettant de rassembler les points de jonction est une chaîne de caractères contenant soit un nom (dans le cas de classes par exemple), soit une signature (dans le cas de méthodes par exemple), soit un patron de recherche, si on veut pouvoir regrouper plusieurs entités. Dans le cas illustré par l'extrait de code 3.1, l'expression est call("% % : :%(...)") qui signifie appel à n'importe quelle méthode de n'importe quelle classe (% : :%), prenant n'importe quels paramètres (...) et ayant n'importe quel type de retour. La coupe transverse est quant à elle appelée *logMethods*.

Un point de jonction correspond à un point du code source où un aspect peut être tissé. En AspectC++, les points de jonction possibles sont les appels et exécutions de méthodes, la lecture ou l'écriture d'un attribut, une classe, une union ou une structure ou l'accès, l'instanciation ou la destruction d'un objet. Dans l'exemple 3.1 le point de jonction est l'appel à n'importe quelle méthode.

Finalement, une méthode d'aspect, définie par le mot clé *advice*, décrit ce que doit faire l'aspect et à quel moment. Ainsi dans l'exemple illustré par l'extrait de code 3.1, la méthode écrit une phrase avant chaque instance de la coupe transverse *logMethods*, soit avant chaque appel à n'importe quelle méthode. En AspectC++, une méthode d'aspect peut être appelée avant (mot clé *before*), après (mot clé *after*) ou à la place (mot clé *around*) du point de

jonction auquel elle est liée. Le développeur a de plus la possibilité de définir dans quel ordre les aspects doivent être utilisés, dans le cas de l'utilisation de plusieurs aspects dans un même programme.

En AspectC++, les aspects sont écrits dans des fichiers *.ah*, ayant une structure similaire à celle des fichiers d'en-tête du C++. Afin de pouvoir être utilisés, les aspects doivent être tissés avec le code source. Pour cela, il existe un tisseur, *ag++*, qui doit être utilisé à la place du compilateur (*g++* par exemple). Les aspects sont ainsi tissés automatiquement à la compilation, ne polluant pas les fichiers C++ du programme. Le choix du tissage à la compilation plutôt qu'à l'exécution vient du fait qu'AspectC++ a été conçu initialement pour des petits programmes embarqués, qui ne doivent pas être surchargés au moment de l'exécution, comme précisé par Spinczyk *et al.* (2002). Cependant, malgré le fait que le tissage se fasse à la compilation, les aspects peuvent quand même avoir accès à des informations disponibles uniquement lors de l'exécution du programme, grâce à une API : la classe *JoinPoint*. Ainsi, l'aspect présenté dans l'extrait de code 3.1 utilise la méthode *signature*() de la classe *JoinPoint* afin d'avoir accès à la signature de la méthode dans laquelle la méthode d'aspect est utilisée.

3.2 GESTION DE LA MÉMOIRE EN C++

La gestion de la mémoire en C++ est délicate. En effet, contrairement á Java, il n'y a pas de ramasse miettes en C++. Ajouter à cela l'utilisation explicite des pointeurs et il devient probable d'avoir des fuites de mémoire, comme l'explique Boehm (2012). Les développeurs doivent donc être attentifs à leur gestion de la mémoire.

Le programme jouet, dont le code source est montré dans l'extrait 3.2, illustre bien ce problème. Dans la fonction *main*, nous instancions un objet de la classe A et deux objets de la classe B. Cependant, au moment de la dés-allocation, seul un objet de chaque classe est détruit, et un objet de la classe B (appelé c) reste en mémoire à la fin du programme. Ceci est une fuite de mémoire évidente, car il n'y a pas de ramasse miettes pour détruire automatiquement l'objet c. Ce programme jouet sera réutilisé par la suite pour réaliser des tests de mémoire.

Parmi d'autres, Sioud (2006) a proposé un ramasse miettes pour le C++. Cependant sa solution est développée en utilisant l'AOP. Elle implémente un simple algorithme de comptage de références, comme décrit par Daper (2012). Le principe est le suivant : à chaque fois que la méthode *new* est appelée, un aspect incrémente un compteur. À l'inverse, à chaque appel à la méthode *delete*, un aspect décrémente le même compteur. À la fin de l'exécution du programme, un aspect inscrit alors la valeur du compteur : si celle-ci est à 0 alors tout va

Extrait de code 3.2 : Code source du programme jouet.

```
#include <stdio.h>
#include <stdlib.h>
#include "A.hpp"
#include "B.hpp"

int main(void) {
    A* a = new A();
    B* b = new B();
    B* c = new B();
    a->foo();
    a->moo();
    c->zoo();
    delete(a);
    b->roo();
    b->koo();
    delete(b);
    c->joo();
    return 0;
}
```

bien, sinon il y a une fuite de mémoire. Ainsi, dans le cas du programme jouet présenté dans l'extrait de code 3.2, le compteur vaudrait 1 à la fin du programme, indiquant la fuite. Cependant une limitation à la solution proposée par Sioud est qu'elle se concentre uniquement sur les méthodes *new* et *delete*, alors que des objets peuvent être créés ou détruits sans utiliser ces fonctions (en utilisant par exemple les fonctions $free$ ou directement les constructeurs), pouvant ainsi faire apparaître des fuites mémoire non détectables par cette approche. Nous verrons par la suite que notre approche permet d'effectuer des tests de mémoire passant outre cette limitation.

3.3 TESTS D'INVARIANT

Les tests d'invariant sont un type de test permettant de s'assurer qu'une propriété d'une méthode, d'une classe ou d'un algorithme est toujours vraie. Ils peuvent être effectués par l'utilisation d'assertions (prédicats indiquant une condition à vérifier, supportés notamment par Java et C++) ou par l'utilisation de méthodes dédiées.

Dans le cas par exemple d'une méthode s'occupant du tri d'un tableau d'entiers, l'invariant peut être le fait qu'à chaque itération de la boucle, les valeurs échangées l'ont été comme il faut (la plus petite se retrouvant avant (ou après) la plus grande). Un test peut ainsi être effectué à chaque itération de la boucle pour vérifier cet invariant.

Les invariants de classe concernent quant à eux les attributs des classes. Un test pourrait ainsi permettre de s'assurer que les attributs ne sont jamais nuls, ou qu'un attribut

représentant une taille soit toujours positif. Ainsi, à chaque fois que l'attribut est modifié, un appel est effectué au test d'invariants qui vérifie que la propriété, la taille est toujours positive, est bien vérifiée.

3.4 BOGUES DE TYPE INTERFÉRENCE

Comme mentionné en section 2.6, les bogues de type interférence sont parmi les plus difficiles à détecter. Un tel bogue se produit lorsqu'au moins deux fils d'exécution d'un même programme tentent d'écrire en même temps dans la même variable. Ce genre d'événement, qui peut être évité grâce à l'utilisation de barrières ou de sémaphores, peut se produire lors de l'injection dans le flux d'exécution du programme d'un délai. En effet, il est courant que les développeurs, pensant pouvoir contrôler leur programme, injectent manuellement des délais dans certains fils d'exécution. Les figures 3.1 et 3.2 illustrent parfaitement ce problème, représentant le flux d'exécution d'un programme.

Dans ce programme, un fil maître crée trois fils esclaves et partage une variable avec eux. Chaque fil, qu'il soit maître ou esclave, peut ainsi lire et écrire dans la variable. Le comportement normal du programme, présenté figure 3.1, est le suivant : chaque fil accède à la variable partagée séquentiellement, récupérant ainsi la dernière version de la variable avant de l'utiliser et de réécrire une nouvelle valeur dans la variable. La figure 3.2 représente quant à elle un comportement anormal du programme. Un délai a été injecté dans le premier fil d'exécution, juste avant que celui-ci n'écrive la nouvelle valeur dans la variable. L'écriture dans la variable par le premier fil intervient alors au même moment que l'écriture effectuée par le deuxième fil, et nous ne savons pas quel fil écrit dans la variable en premier : nous avons affaire à un bogue de type interférence. Le deuxième fil a lu une variable erronée (elle aurait dûe être modifiée par le premier fil mais ne l'a pas été), et le troisième fil d'exécution, qui s'attend à lire le résultat du deuxième fil, lira lui aussi une variable erronée (soit car elle correspondra au résultat du premier fil, soit car elle correspondra au résultat du deuxième fil avec une entrée erronée). Nous verrons dans la section 5.5 que notre approche permet de fournir les données d'entrée nécessaires à une méthode permettant la découverte de ce type de bogue.

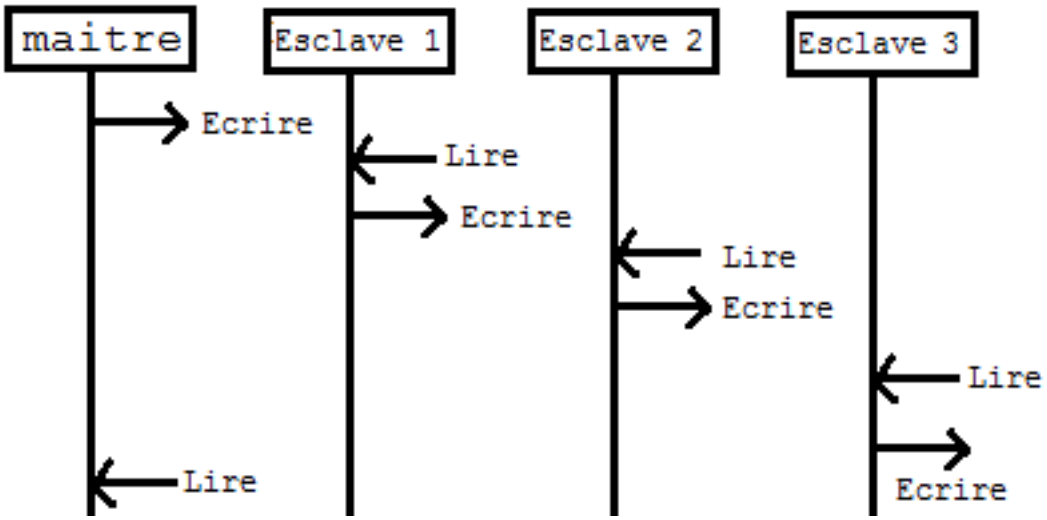

Figure 3.1 : Exemple de comportement normal.

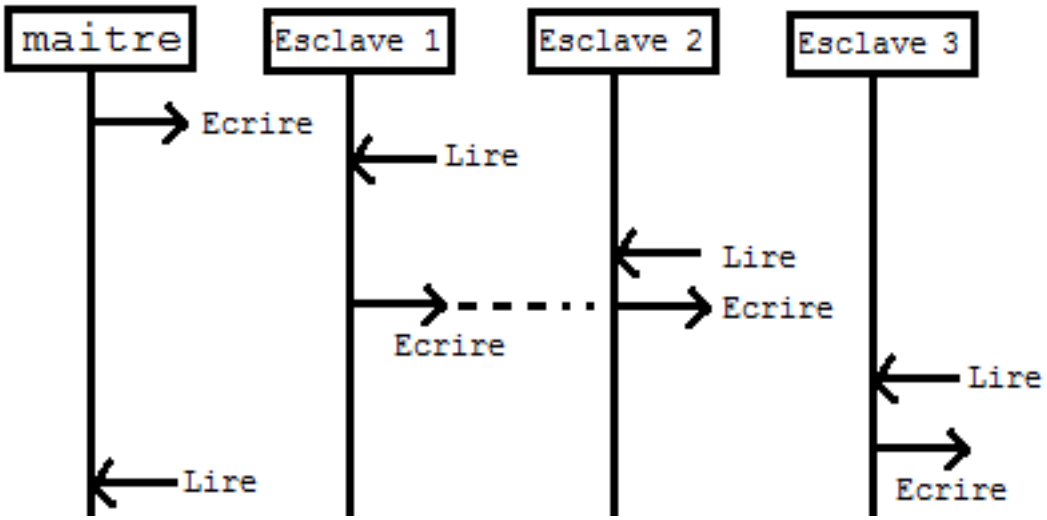

Figure 3.2 : Exemple de comportement anormal.

Chapitre 4

APPROCHE

Ce chapitre présente les détails de notre approche. La section 4.1 décrit le DSL que nous proposons, chaque sous section étant consacrée plus précisement à un type d'aspect que le DSL permet de construire. La section 4.2 introduit quant à elle ACRE, le logiciel développé afin de lire le DSL et de générer automatiquement les aspects à partir de celui-ci.

4.1 DSL

Afin de pouvoir créer automatiquement les aspects utilisés pour tester les logiciels, nous devons pouvoir trouver les informations nécessaires à leur création dans le code du programme. En Java les programmeurs peuvent utiliser les annotations pour cela. Cependant, ces dernières n'existant pas en C++, nous avons développé un DSL (langage dédié) permettant aux testeurs de décrire les tests dont ils ont besoin. Comme nous voulions pouvoir insérer ce DSL dans le code source du programme sans pour autant rendre ce dernier non compilable, nous avons fait en sorte que notre DSL ne modifie pas le comportement du programme. Le choix d'introduire la description des aspects dans le code source du programme est fait pour faciliter le travail des testeurs : ils peuvent ainsi écrire les tests qu'ils veulent effectuer comme s'ils écrivaient des commentaires et ce lors de l'écriture du code source.

Notre DSL devait répondre aux requis suivant : il nous fallait tout d'abord pouvoir identifier facilement les lignes de codes DSL au milieu des lignes de code C++. Nous avions aussi besoin de connaître le type d'aspect à créer, ainsi que la classe ou méthode, suivant le type, à laquelle l'aspect se rattachait, et si celui-ci devait s'exécuter avant ou après la dite méthode. Finalement, pour les tests d'invariant, notre DSL devait nous permettre de savoir quel invariant tester, et comment l'obtenir. Pour répondre à toutes ces exigences, notre DSL possède les caractéristiques décrites dans les sections suivantes.

Pour que les lignes de code du DSL soient identifiables, autant par leur rédacteur que par ACRE, nous avons décidé qu'elles devaient commencer par la séquence ////. Ce symbole satisfait aussi la contrainte de garder le code compilable, vu qu'il n'est pas utilisé en C++. Une règle que nous avons définie est que toutes les lignes servant à définir un même aspect doivent être consécutives et au moins une ligne de code non DSL doit séparer deux définitions d'aspects. Les définitions ont ensuite deux lignes communes avant d'être différenciées suivant le type d'aspect voulu. Ainsi, pour chaque aspect, la première ligne contient le nom de l'aspect

à créer, défini par la ligne :

//// name : AspectName

La deuxième ligne contient le type d'aspect à construire, via la ligne :

//// type : AspectType

Les sous-sections suivantes présentent plus en détail les quatre types d'aspect actuellement supportés par notre approche, et nommés respectivement *counter*, *logging*, *timing* et *checking*. Le tableau 4.1 synthétise la syntaxe du DSL.

4.1.1 *COUNTER*

Un aspect de type *counter* fait trois choses. Tout d'abord, il ajoute un attribut statique à la classe indiquée dans sa description. Il équipe ensuite les constructeurs et destructeurs de cette classe pour qu'ils incrémentent et décrémentent respectivement l'attribut ajouté. Finalement, à la fin de l'exécution du programme, l'aspect affiche la valeur de l'attribut, indiquant, si celle-ci est supérieure à zéro, que le programme a éventuellement une fuite mémoire dûe à cette classe. La principale différence entre un aspect de type *counter* et l'approche proposée par Sioud est le fait que les aspects de type *counter* modifient les constructeurs et les destructeurs de la classe, et ne se concentrent donc pas uniquement sur les méthodes *new* et *delete*. Pour construire ce type d'aspect, en plus des deux lignes obligatoires, deux autres lignes optionnelles peuvent être indiquées. La première, sous la forme

//// className : ClassName

sert à indiquer le nom de la classe dans laquelle l'attribut doit être ajouté. Si jamais cette ligne n'est pas présente, ACRE utilisera le nom du fichier dans lequel la définition se trouve comme nom de classe. Si plusieurs classes sont définies dans un même fichier, cette ligne est alors obligatoire pour différentier les différents compteur à créer. Ainsi, si les lignes de DSL écrites dans l'extrait de code 4.1 se trouvent dans le fichier *Eval_Point.hpp*, l'attribut sera ajouté dans la classe *Eval_Point*.

Extrait de code 4.1 : Code utilisé pour générer un aspect de type *counter*.

```
//// name: EVPCG
//// type: counter
```

La deuxième ligne optionnelle pour construire un aspect de type *counter* est de la forme suivante :

//// namespace : NamespaceName

Comme son nom l'indique elle sert à préciser l'espace de nommage (*namespace*) dans lequel

se trouve la classe choisie. En effet, pour pouvoir trouver une classe, AspectC++ a besoin de connaître son espace de nommage. Ainsi si la classe *Eval_Point* est dans l'espace de nommage NOMAD, mais que lors de la description de l'aspect ceci n'est pas mentionné, AspectC++ ne pourra pas appliquer l'aspect à la classe car il ne la trouvera pas. Si jamais cette ligne n'est pas présente dans la description de l'aspect, ACRE va chercher dans le fichier contenant la description si jamais une définition d'un espace de nommage est présente ou non. S'il n'en trouve pas, l'espace de nommage reste indéfini dans l'aspect et AspectC++ risque donc de ne pas trouver la classe à laquelle ajouter l'attribut. Cette ligne n'est pas obligatoire car toutes les classes ne font pas forcément partie d'un espace de nommage.

Les lignes de codes présentées dans l'extrait 4.1 ont servi à créer un aspect permettant la découverte d'une fuite mémoire dans NOMAD, comme nous le verrons dans la partie 5.

4.1.2 *LOGGING*

Un aspect de type *logging* permet d'obtenir une trace du programme dans lequel il est tissé en permettant au développeur d'afficher une phrase à chaque fois que son programme entre ou sort d'une méthode, comme présenté dans la section 3.1. Pour décrire ce type d'aspect, la troisième ligne de la description doit être de la forme

//// method && call / execution && MethodName

Elle sert à spécifier quelle méthode est concernée par l'aspect et si l'aspect est appelé lors de l'exécution ou lors de l'appel de cette méthode. L'appel à la méthode *bla* est le moment ou une autre méthode foo utilise *bla*, tandis que l'exécution d'une méthode est le moment où le corps de la méthode s'exécute réellement. L'attribut $MethodName$ doit contenir la signature de la méthode concernée. Si l'aspect concerne plusieurs méthodes alors l'attribut $MethodName$ doit être un patron de recherche similaire à une signature. Il doit ainsi contenir le type de retour des méthodes, le nom de la classe contenant ces méthodes (éventuellement précédé de l'espace de nommage auquel elle appartient), ainsi que la liste des arguments pris en paramètre, si elles en prennent. Ainsi, les méthodes concernées par l'aspect décrit dans les lignes de code DSL de l'extrait 4.2 sont toutes les méthodes de l'espace de nommage NOMAD.

Extrait de code 4.2 : Code utilisé pour générer un aspect de type *logging*.

```
//// name: loggingAspect
//// type: logging
//// methods && call && % NOMAD::%(...)
//// when: before
//// when: after
```

Une quatrième ligne optionnelle pour décrire un aspect de type *logging* est de la forme

//// when : before / after

Elle sert à indiquer si l'aspect doit écrire une phrase avant ou après l'appel ou l'exécution de la méthode concernée. Une méthode d'aspect est créée pour chaque ligne de ce type présente dans la description de l'aspect. La description présentée dans l'extrait 4.2 créera ainsi un aspect contenant deux méthodes d'aspect : une écrivant une phrase avant chaque appel de chaque méthode de l'espace de nommage NOMAD, l'autre faisant de même après chaque appel. Pour l'instant l'utilisateur n'a pas la possibilité de particulariser ce qui est écrit.

4.1.3 *TIMING*

Un aspect de type *timing* permet d'obtenir les temps d'accès en lecture et/ou en écriture d'un attribut d'une classe accessible par un accesseur (*getter*) et une méthode de réglage (*setter*). Pour cela, la troisième ligne de la description doit être de la forme

//// time : second / nanosecond

Elle permet au développeur de préciser s'il veut les temps d'accès en secondes ou en nanosecondes. Par la suite la description doit contenir au moins une des trois lignes suivantes :

//// attribute : AttributeName [&& set && get]

//// set : SetterSignature

//// Get : GetterSignature

Ces lignes permettent de connaître l'attribut concerné par l'aspect et si on veut les temps de lecture, d'écriture ou les deux. Dans la ligne

//// attribute : AttributeName [&& set && get]

seul le nom de l'attribut est renseigné, ainsi que si on veut ses accès en écriture (*set*) et/ou en lecture (*get*). Lors de la création de l'aspect, ACRE va considérer que l'attribut appartient à la classe portant le nom du fichier dans lequel la description se trouve. Pour l'espace de nommage ACRE procède de la même manière que pour un aspect de type *counter* : il cherche une déclaration d'espace de nommage dans le fichier. Enfin, pour les getters et setters, ACRE considère qu'ils sont de la forme $set_AttributeName(...)$ et $get_AttributeName(...)$.

Si les méthodes ne sont pas de cette forme, alors cette ligne de devra pas être utilisée. Ainsi, pour les lignes de code DSL présentées dans l'extrait 4.3, et écrites dans le fichier $Mesh.hpp$ contenant la description de l'espace de nommage NOMAD, la méthode concernée sera

% NOMAD : :Mesh : :set_mesh_index(...)

Pour les lignes de type

//// set : SetterSignature

et

//// get : GetterSignature

la signature de la méthode doit être complète, c'est à dire contenir le type de retour, la classe (et l'espace de nommage), le nom de la méthode et ses éventuels paramètres (comme pour l'attribut $MethodName$ dans la description d'un aspect de type *logging*).

Extrait de code 4.3 : Code utilisé pour générer un aspect de type *timing*.

```
//// name: TimingAspect
//// type: timing
//// time : nanosecond
//// attribute : mesh_index && set
```

4.1.4 *CHECKING*

Un aspect de type *checking* est un aspect ne contenant qu'une méthode d'aspect et qui peut faire tout ce que le développeur veut, tant que cela reste réalisable par un aspect. Pour cela, la troisième ligne dans la description doit être de la forme

//// before / after / around : [return_type__][className : :]MethodName[(arguments)]

Elle sert à définir quand la méthode d'aspect doit être utilisée (avant, pendant ou après quelle(s) méthode(s)). Comme pour les autres définitions d'aspects, l'attribut $className$ peut contenir l'espace de nommage de la classe, sous la forme

`namespace::className`.

Par la suite, le développeur décrit ce qu'il veut que sa méthode d'aspect fasse. Pour cela neuf types de ligne de code DSL sont disponibles. La première d'entre elles permet la création d'une variable et se présente sous la forme suivante :

//// create : type && VariableName

Si le développeur veut utiliser un constructeur, par exemple pour déclarer une variable $A\ a(x)$, alors l'attribut $VariableName$ doit contenir le nom de la variable, suivi par des parenthèses contenant les éventuels paramètres. Ainsi, pour créer la ligne précédente, la ligne de code DSL requise est :

//// create : A && a(x)

Pour initialiser une variable, l'attribut $VariableName$ doit contenir toutes les informations requises pour l'initialisation. Ainsi pour obtenir $int\ i = 0$, il faut écrire

//// create : int && i = 0.

La seconde ligne disponible permet de donner une valeur à une variable et se présente sous la forme :

////store : type && howToGet

L'attribut *howToGet* doit être de la forme $a.b()$ ou $a \to b()$, où $b()$ est une méthode retournant

la variable, par exemple un *getter*. Ainsi pour stocker la valeur retournée par la méthode $get_n()$ de l'objet a à une variable, on doit écrire la ligne :

//// store : int && $a \rightarrow get_n()$.

Les six autres lignes disponibles sont les suivantes :

//// do : if(condition)

et

//// do : endIf

qui permettent d'ouvrir et de fermer un bloc *if*

//// do : while(condition)

et

//// do : endWhile

qui permettent la construction d'un bloc *while*, et

//// do : for-variableName-begin-end

et

//// do : endFor

que permettent de créer un bloc *for*. Finalement, la neuvième ligne disponible est la suivante :

//// do : line

Elle permet au développeur d'écrire une ligne qui sera écrite telle quelle lors de la création de l'aspect par ACRE. L'extrait de code 4.4 montre un exemple de description d'un aspect de type *checking* contenant presque toutes les lignes disponibles. L'aspect créé par cette description sera discuté en section 5.

Extrait de code 4.4 : Code utilisé pour générer un aspect de type *checking*.

```
//// name: Deltas
//// type: checking
//// after: iteration
//// store: NOMAD::Signature * && this->_p.get_signature()
//// store: int && signature->get_n()
//// store: NOMAD::Mesh && signature->mesh()
//// store: int && mesh.get_mesh_index()
//// create: NOMAD::Point && delta_m(n)
//// create: NOMAD::Point && delta_p(n)
//// do: mesh.get_delta_p(delta_p, mesh_index)
//// do: mesh.get_delta_m(delta_m, mesh_index)
//// do: for-i-0-n
//// do: if (delta_m[i] > delta_p[i])
//// do: printf("Error")
//// do: endIf
//// do: endFor
```

Tableau 4.1 : Synthèse du DSL.

Type	Ligne	Description
Tous	//// name : AspectName	Nom de l'aspect généré
	//// type : AspectType	Type de l'aspect généré
Counter	//// className : ClassName	Classe concernée par l'aspect
	//// namespace : NamespaceName	Espace de nommage contenant la classe
Logging	//// method && call / execution && MethodName	Méthode concernée par l'aspect
	//// when : before / after	L'aspect s'exécute avant ou après la méthode
Timming	//// time : second / nanosecond	Unité de temps
	//// attribute : AttributeName [&& set && get]	Attribut concerné par l'aspect
	//// set : SetterSignature	Setter de l'attribut concerné
	//// get : GetterSignature	Getter de l'attribut concerné
Checking	//// before / after / around : [return_type__] [className : :]MethodName[(arguments)]	L'aspect s'effectue avant ou après la méthode concernée
	//// create : type && VariableName	Crée une variable de type type
	//// store : type && howToGet	Donne une valeur à une variable
	//// do : if(condition)	Crée un bloc if
	//// do : endIf	
	//// do : while(condition)	Crée un bloc while
	//// do : endWhile	
	//// do : for-variableName-begin-end	Crée un bloc for
	//// do : endFor	
	//// do : line	Écrit la ligne line dans l'aspect

4.2 ACRE

Une fois les aspects décrits dans le code source du programme, nous utilisons ACRE pour lire les descriptions et générer les aspects correspondants. ACRE, pour **A**utomatic aspe**C**t c**RE**ator, est un analyseur qui construit automatiquement les aspects correspondant aux descriptions écrites en ligne de code DSL. Il est écrit en Java et est disponible en téléchargement depuis l'Internet [1] sous licence GPL.

Pour fonctionner, ACRE prend en entrée un dossier contenant du code source C++ (fichiers *.hpp* et / ou *.cpp*), compilable ou non. ACRE parcourt ensuite ces fichiers à la recherche de descriptions d'aspects. Pour chaque définition trouvée, ACRE regarde le type d'aspect et appelle le sous-analyseur (*parser*) correspondant. Ce dernier analyse la description, crée le fichier *.ah* et le remplit avec l'aspect correspondant, écrit en AspectC++. Si un problème survient durant l'analyse de la description, à cause par exemple d'une ligne erronée, alors ACRE ne créé pas l'aspect correspondant et en informe l'utilisateur via un message. Il continue cependant d'analyser les définitions restantes. Afin que l'aspect généré puisse compiler, ACRE lui inclut automatiquement les bibliothèques standards du C, à savoir *stdio* et *stdlib*. Il inclut aussi le fichier contenant les lignes de code DSL lues, l'aspect ayant ainsi accès aux même informations que la classe ou la méthode à laquelle il est lié. Finalement, il traduit aussi les mots clés de C++ en mots clés d'AspectC++. Ainsi, *this*, qui n'existe pas en AspectC++, devient $tjp \rightarrow that()$, *tjp* signifiant *this join point*, ce point de jonction. ACRE effectue aussi quelques actions particulières pour certains types, comme expliqué ci-après.

Pour les aspect de type *checking*, nous avons vu en section 4.1.4, que l'action *store* est utilisée pour donner une valeur à une variable. Cependant le nom de la variable recevant la valeur n'est pas mentionné dans la ligne DSL, qui, pour rappel, est de la forme

////store : type && howToGet

En effet, le nom de la variable est attribué par ACRE au moment de la création de l'aspect. Ainsi, si la méthode servant à obtenir la valeur est un *getter*, sous la forme *get_*, alors la variable reçoit le nom de l'attribut concerné. Par exemple, si la ligne de code DSL lue dans la description est

//// store : int && this→ get_size()

alors la ligne générée dans l'aspect sera :

int size = tjp→ that()→ get_size() ;

Si, cependant, la méthode n'est pas de la forme get_, alors la variable aura pour nom le nom de la méthode elle-même :

//// store : int && a.size()

1. `http://web.soccerlab.polymtl.ca/~ducloset/ACRE`

générera ainsi

```
int size = a.size();
```

Dans ce cas l'utilisateur doit faire attention à ne pas appeler deux fois la même méthode, sinon deux variables posséderons le même nom et l'aspect ne pourra fonctionner.

Pour les aspects de type *counter*, un changement est requis dans le code source du programme, contrairement aux autres types d'aspects. En effet, il faut initialiser l'attribut représentant le compteur. Pour cela ACRE écrit, en commentaire dans l'aspect, les deux lignes requises pour cette initialisation. Il revient ensuite à l'auteur de copier ces deux lignes dans le code source de son programme, ACRE le lui rappelant dans ses sorties. L'écriture de ces deux lignes aurait pu être automatique, mais nous avons préféré laisser le soin au développeur de modifier lui même son propre code source et de ne pas lui imposer nos choix. De plus, si les deux lignes restent écrites dans le code même pour une version où l'aspect ne sera pas tissé, alors le code ne compilera pas. Il est donc important que le développeur sache où sont ces lignes. L'extrait de code 5.4 fourni un exemple d'aspect de type *counter* généré automatiquement par ACRE et qui contient ces deux lignes.

Actuellement, ACRE ne peut gérer que quatre types d'aspects, décrits précédemment. Cependant, la façon dont ACRE est construit permet l'ajout simple de nouveaux types d'aspect. En effet, lors de la lecture d'une description d'aspect, ACRE repère le type de l'aspect à créer et transfère ensuite la gestion de cette description et l'écriture de l'aspect correspondant au sous-analyseur adéquat. L'ajout de la gestion d'un nouveau type d'aspect se fait donc via l'ajout d'un nouveau sous-analyseur spécifique. Afin que cet ajout soit le plus simple possible et ne demande pas de modifications dans ACRE, nous fournissons une interface indiquant au développeur les signatures des deux méthodes qu'il doit implémenter pour avoir un sous-analyseur fonctionnel. Ces deux méthodes servent respectivement à lire la description de l'aspect et à générer l'aspect.

Chapitre 5

ÉTUDE EMPIRIQUE

La section 5.1 expose les différentes hypothèses, tandis que la section 5.2 introduit NOMAD, le logiciel testé. Les sections 5.3, 5.4 et 5.5 traitent respectivement des résultats obtenus pour les hypothèses 1, 2 et 3. La section 5.6 discute quant à elle des résultats présentés dans les sections qui la précèdent.

5.1 HYPOTHÈSES DE RECHERCHE

Le but de notre étude est d'utiliser l'AOP afin de réaliser des tests de mémoire, d'invariant et d'interférence sur des programmes écrits en C++, tel NOMAD. Pour cela nous avons tout d'abord dû vérifier si cela était possible, ce qui nous a conduit à la première hypothèse :

Hypothèse 1 : *L'AOP peut être utilisée efficacement pour tester des programmes C++.*

La revue de littérature nous ayant montré que cela était possible pour les programmes écrits en Java et pour les programmes C++ embarqués, cette hypothèse cherche à nous assurer et montrer qu'effectivement l'AOP permet d'effectuer différents types de tests dans des programmes C++ et notamment dans NOMAD.

Une fois cette hypothèse vérifiée, notre but est de pouvoir générer automatiquement les aspects permettant de tester les programmes, ce qui nous amène à la deuxième hypothèse :

Hypothèse 2 : *Des aspects peuvent être générés automatiquement depuis le code source de programmes C++.*

Cette hypothèse est utilisée pour savoir si une automatisation est possible dans la génération des aspects, notamment en utilisant le DSL que nous proposons (voir section 4.1) et ACRE (voir section 4.2).

Après avoir validé ces deux hypothèse et pu trouver des bogues dans NOMAD, notamment des fuites de mémoire, nous nous sommes alors consacrés aux bogues de type interférence et avons voulu voir si notre approche peut être utilisée pour aider à les détecter. Nous avons pour cela formulé une troisième hypothèse :

Hypothèse 3 : *Les aspects ainsi générés peuvent être utilisés pour aider à la réalisation de tests d'interférence.*

Le critère de succès considéré pour notre approche est de pouvoir réaliser au moins un test de chaque type (mémoire, invariant et interférence) dans NOMAD grâce à un aspect généré automatiquement par ACRE, et que ces tests trouvent des bogues. Ces bogues ne doivent cependant pas être trop spécifique à NOMAD pour que l'approche puisse être considérée comme généralisable. L'oracle utilisé pour le bogue reporté dans les résultats est un rapport d'utilisateur concernant une fuite mémoire dans NOMAD. Pour les résultats concernant le programme jouet, l'oracle est le fait que la fuite mémoire a été introduite sciemment.

Les utilisateurs visés par notre approche sont les développeurs et les testeurs qui veulent tester des programmes C++ sans avoir besoin de modifier le comportement du code source de leur programme. Notre approche peut aussi être utile aux personnes voulant tester leur programme mais qui ne possèdent pas de connaissances spécifiques en test de logiciel ni en programmation orientée aspect.

5.2 NOMAD

NOMAD, développé en C++ par Le Digabel (2011), est un logiciel d'optimisation de boîte noire pouvant être utilisé sur plusieurs systèmes d'exploitation (Windows, Linux et Mac Os). C'est un logiciel utilisé à la fois pour la recherche et par des industriels, tels Airbus, Boeing, Exxon Mobil ou Hydro-Québec. Nous espérons qu'obtenir des résultats sur un tel logiciel (écrit en C++ et utilisé en recherche et en industrie) signifie que nous pourront en obtenir sur tous les logiciels C++. NOMAD est disponible en téléchargement depuis l'Internet [1] sous la licence LGPL, et a été téléchargé près de 3 000 fois depuis 2008, comme le montre la figure 5.1.

NOMAD a été conçu pour résoudre des problèmes difficiles d'optimisation de fonctions de type boîte noire impliquant la minimisation d'une fonction objectif sous contraintes. La fonction à optimiser est passée en paramètre du programme sous la forme d'un exécutable, renseigné dans le fichier de paramètres fourni à NOMAD.

Une fonction de type boîte noire est une fonction qui peut être bruitée, chère à calculer et qui peut échouer sur l'évaluation de points à priori réalisables. De plus, on ne connaît de ces fonctions que les données d'entrée et de sortie, et nous ne pouvons avoir accès à leur dérivées ou autres caractéristiques utiles. Ces fonctions se retrouvent souvent en ingénierie, par exemple pour calculer la forme d'une aile d'avion. Pour optimiser de telles fonctions, NOMAD est basé sur l'algorithme MADS, pour Mesh Adaptive Direct Search, proposé par Audet

1. `http://www.gerad.ca/nomad`

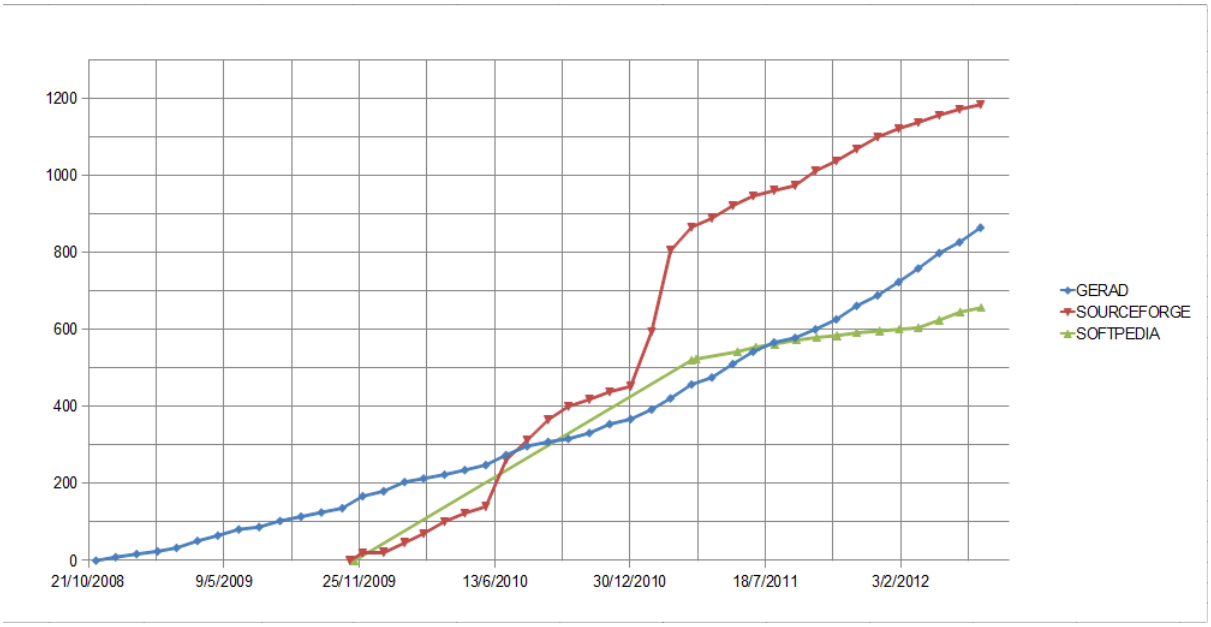

Figure 5.1 : Courbe des téléchargements de NOMAD depuis 2008, selon 3 sources.

et Dennis, Jr. (2006). Le problème d'optimisation se présente sous la forme

$$\min_{x \in \Omega} f(x)$$

dans le cas d'un problème avec un seul objectif ou

$$\min_{x \in \Omega} \ (f_1(x), f_2(x))$$

pour un problème d'optimisation biobjectif, où Ω est l'ensemble des points réalisables, et f, f_1 et f_2 sont les fonctions de type boîte noire à optimiser.

L'algorithme MADS, dont une version simplifiée est présentée dans le tableau 5.1, fonctionne de la manière suivante, dans le cadre d'un problème avec un seul objectif : à partir d'un point x_k, on cherche un minimum de f en évaluant quelques points situés sur un treillis (ou *mesh*) de taille Δ_k^m. L'algorithme se base sur des directions denses dans l'espace afin de sélectionner des points candidats. Ces candidats constituent l'ensemble

$$P_k = \{x_k + \Delta_k^m d : d \in D_k\}$$

où D_k est un ensemble de directions. Les points ainsi sélectionnés se situent à une distance bornée par un paramètre Δ_k^p (non représenté ici) du point x_k. La figure 5.2 illustre ceci. Les

lignes minces représentent le treillis de taille Δ_k^m et les lignes épaissent les points à distance Δ_k^p du point x_k. L'étape recherche et sonde se déroule comme suit : à chaque itération, une recherche locale, basée sur des directions denses, est d'abord effectuée. Il s'agit de la phase "sonde". La deuxième phase consiste en une recherche globale. Cette partie est plus flexible et permet de générer des points sur le treillis. Ces points peuvent par ailleurs être fournis par l'utilisateur si celui-ci a une certaine connaissance du problème.

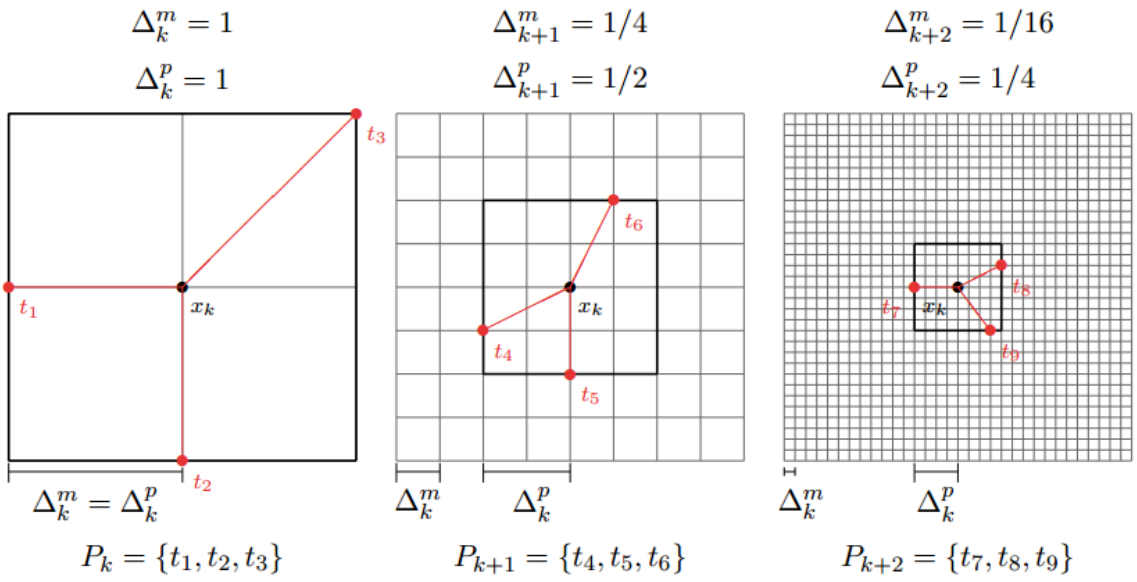

Figure 5.2 : Exemple de différentes configuration de treillis, tiré de Le Digabel (2011) (Reproduction avec autorisation).

Lors de cette recherche, si un point x_{k+1} parmi ceux évalués vérifie $f(x_{k+1}) < f(x_k)$, alors la taille Δ_k^m du treillis augmente et l'algorithme recommence avec x_{k+1} à la place de x_k, et ce jusqu'à ce que le nombre d'évaluations maximal fixé par l'utilisateur soit atteint. Si, par contre, aucun point n'est meilleur que x_k, alors l'algorithme diminue la taille du treillis et recommence la recherche centrée autour de x_k. Plus de détails concernant l'algorithme MADS sont disponibles dans l'article de Audet et Dennis, Jr. (2006).

NOMAD permet une grande flexibilité dans l'utilisation de MADS, autorisant l'ajustement manuel de près de 100 paramètres, allant du nombre d'évaluations maximal à la précision des réels utilisés. Le tableau 5.2, tiré du manuel d'utilisation de NOMAD (Le Digabel et Tribes (2011)), est un exemple de fichier contenant des paramètres d'entrée pour NOMAD.

Le tableau 5.3, lui aussi tiré du manuel d'utilisation, représente un exemple de sortie générée par NOMAD pour les paramètres donnés dans le tableau 5.2.

NOMAD possède plusieurs type de sorties, suivant le degré de détail demandé par l'utilisateur. Le tableau 5.3 représente la sortie standard, où on peut voir le nombre d'évaluations ef-

Tableau 5.1 : Algorithme MADS simplifié, inspiré de Le Digabel (2011).

- Initialisation : initialiser Δ_0^m et Δ_0^p, et mettre le compteur d'itérations k à 0.
- Étape recherche et sonde : chercher un point améliorant x_{k+1} sur le treillis M_k
- Mises à jour : Mettre à jour Δ_{k+1}^m, Δ_{k+1}^p ; $k \leftarrow k+1$
  recommencer l'étape recherche et sonde.

fectuées, la meilleure solution non-réalisable trouvée et la meilleure solution réalisable trouvée, chacune associée à son coût (f).

Tableau 5.2 : Exemple de paramètres d'entrée pour NOMAD, inspiré de Le Digabel et Tribes (2011).

DIMENSION	5	# nombre de variable
BB_EXE	bb.exe	# bb.exe est un programme qui prend
BB_OUTPUT_TYPE	OBJ PB EB	# en argument le nom d'un fichier texte # contenant 5 valeurs, et qui affiche 3 # valeurs correspondantes aux valeurs # de la fonction objective (OBJ), et 2 # valeurs de contrainte g1 et g2 avec # $g1 \leq 0$ et $g2 \leq 0$; PB et EB # correspondent aux contraintes qui # sont traitées par l'approche # Progressive et Extreme Barrier (toutes # les option de traitement des # contraintes sont décrites dans la # liste détaillée des paramètres)
X0	(0 0 0 0 0)	# point de départ
LOWER_BOUND	* -6	# toutes les variables sont ≥ 6
UPER_BOUND	(5 6 7 - -)	# $x_1 \leq 5$, $x_2 \leq 6$, $x_3 \leq 7$ # x_4 and x_5 n'ont pas de bornes supérieures
MAX_BB_EVAL	100	# l'algorithme se terminera quand # 100 évaluations de la boîte noire # auront été effectuées

Tableau 5.3 : Exemple de sortie générée par NOMAD, inspiré de Le Digabel et Tribes (2011).

```
NOMAD - version 3.5.1 - www.gerad.ca/nomad

Copyright (C) 2001-2012 {
        Mark A. Abramson  - The Boeing Company
        Charles Audet     - Ecole Polytechnique de Montreal
        Gilles Couture    - Ecole Polytechnique de Montreal
        John E. Dennis, Jr.   - Rice University
        Sebastien Le Digabel - Ecole Polytechnique de Montreal
        Christophe Tribes    -  Ecole Polytechnique de Montreal
}

Funded in part by AFOSR and Exxon Mobil.

License     : '\$NOMAD\_HOME/src/lgpl.txt'
User guide : '\$NOMAD\_HOME/doc/user\_guide.pdf'
Examples  : '\$NOMAD\_HOME/examples'
Tools       : '\$NOMAD\_HOME/tools'

Please report bugs to nomad@gerad.ca

MADS run {

BBE ( SOL ) OBJ

   2 ( 1.100000000 0.000000000 0.000000000 0.000000000 0.000000000 ) 275.228100000
   3 ( 4.400000000 0.000000000 0.000000000 0.000000000 0.000000000 ) 0.000000000
100 ( 4.400000000 0.000000000 0.000000000 0.00000000 0.000000000 ) 0.000000000

} end of run (max number of blackbox evaluations)

blackbox evaluations    : 100
best infeasible solution : ( 4.4 -2.4 0 0 0 ) h=1.12 f=0
best feasible solution    : ( 4.4 0 0 0 0 ) h=0 f=0
```

5.3 VALIDATION DE L'HYPOTHÈSE 1

La première hypothèse, formulée *l'AOP peut être utilisée efficacement pour tester des programmes C++ ?*, doit nous convaincre que des bogues peuvent être trouvés dans des programmes C++ grâce à des aspects. Pour vérifier cette hypothèse, nous avons procédé en deux étapes. Nous nous sommes tout d'abord concentrés sur le programme jouet, présenté en section 3.2 et par l'extrait de code 3.2, avant de tester notre approche sur un programme réel, NOMAD.

Le programme jouet ayant une fuite mémoire évidente (un objet instancié n'est pas libéré), nous avons construit un aspect permettant de nous l'indiquer. L'aspect dédié en question est présenté dans l'extrait de code 5.1. Il a le fonctionnement suivant : chaque fois qu'un constructeur de la classe B (ligne 11) est appelé, la variable statique *count* est incrémentée (ligne 16). De même, chaque fois qu'un destructeur de la classe B est appelé, la même variable est décrémentée (ligne 17). Finalement, l'aspect écrit à la fin du programme (ligne 21) la valeur de la variable *count* (ligne 22). Nous initialisons ensuite la variable *count* à 0, en rajoutant dans le fichier *B.cpp*, contenant le corps des méthodes de la classe B, les lignes écrites dans l'extrait 5.2. Une fois l'aspect tissé avec le code source, nous exécutons le programme jouet, et, comme espéré, la phrase *Final count of B : 1* est affichée. L'utilisation de l'AOP nous a donc permis de découvrir la fuite mémoire.

Le résultat obtenu sur le programme jouet étant encourageant, nous considérons maintenant un programme réel, NOMAD. L'équipe de développement de NOMAD a reçu un relevé de bogues concernant une fuite mémoire lorsque NOMAD tourne sous Windows pour un problème particulier. Nous avons donc décidé de chercher cette fuite en utilisant l'AOP. Nous modifions pour cela l'aspect utilisé pour trouver la fuite mémoire dans le programme jouet, remplaçant la classe B par un ensemble de classes de NOMAD, les plus susceptibles selon nous d'être à l'origine de la fuite (c'est-à-dire les classes sur lesquelles pointent le plus de pointeurs ou les classes les plus utilisées. Ces classes nous ont été communiquées par l'équipe de développement de NOMAD). Après avoir initialisé le compteur (l'attribut de l'aspect) et tissé l'aspect (en modifiant le makefile pour utiliser ag++ au lieu de g++), nous exécutons NOMAD sur le problème indiqué dans le relevé de bogues. Le résultat obtenu est qu'il y a, parmi tous les objets des classes concernées, 74 objets instanciés qui ne sont jamais détruits. Cela ne nous permet cependant pas de savoir quelle(s) classe(s) est (sont) concernée(s), l'attribut affiché étant un attribut unique pour toutes les classes testées. Afin d'être sûr de trouver toutes les classes concernées par la fuite mémoire, nous créons un aspect par classe, réitérons les étapes et trouvons que, finalement, seule la classe *Eval_Point* est concernée,

Extrait de code 5.1 : Aspect permettant de trouver la fuite mémoire dans le programme jouet section 3.2.

```
1  #ifndef _MYTOYCOUNTER_AH_
2  #define _MYTOYCOUNTER_AH_
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  aspect MyToyCounter {
8     public:
9        static int count;
10
11       pointcut counted() = "B";
12
13       advice counted() : slice struct {
14          class Helper {
15             public:
16                Helper() {MyToyCounter::count++;}
17                ~Helper(){MyToyCounter::count--;}
18          } counter;
19       };
20
21       advice execution("% main(...)") : after() {
22          printf("Final count of B: %d\n", count);
23       }
24 };
25
26 #endif
```

Extrait de code 5.2 : Lignes permettant l'initialisation de la variable de l'aspect MyToyCounter.

```
#include "MyToyCounter.ah"
int MyToyCounter::count = 0;
```

comme le montre la figure 5.3.

Ce résultat nous permet de valider notre première hypothèse : l'AOP peut effectivement être utilisée pour tester des logiciels réels écrits en C++. Ce résultat ne concerne cependant que les tests de mémoire, et tous les aspect utilisés pour obtenir cette réponse ont été créés manuellement.

5.4 VALIDATION DE L'HYPOTHÈSE 2

La deuxième hypothèse, formulée *des aspects peuvent être générés automatiquement depuis le code source de programmes C++ ?*, doit nous convaincre qu'une automatisation de la génération des aspects est possible, grâce au DSL et à ACRE, présentés respectivement dans

```
         979    (       0.3975788348 8.012365157 0.3625952099 0.2312161398
        1000    (       0.3975788348 8.012365157 0.3625952099 0.2312161398
} end of run (max number of blackbox evaluations)

blackbox evaluations    : 1000
best feasible solution  : ( 0.3975788348 8.012365157 0.3625952099 0.2312161
19424 0.01103221486 1.478404609 ) h=0 f=0.5214127441
Final count of Eval_Point : 74
Memory leak !!
```

Figure 5.3 : Sortie de NOMAD avec un aspect de type *counter* tissé.

les sections 4.1 et 4.2. Pour valider cette hypothèse, nous procédons de la même manière que pour la précédente, nous concentrant tout d'abord sur le programme jouet (présenté dans l'extrait de code 3.2) avant de considérer NOMAD.

Afin de générer automatiquement un aspect nous permettant de trouver la fuite mémoire présente dans le programme jouet, nous utilisons notre DSL et ACRE. Nous écrivons ainsi les trois lignes montrées dans l'extrait 5.3 dans le fichier *B.hpp*, et exécutons ACRE sur le dossier contenant le code source du programme jouet. Un aspect, sensiblement identique à celui présenté dans l'extrait 5.1, est ainsi créé. Nous le tissons donc avec le code source du programme jouet, enlevant tous les aspects créés manuellement précédemment, et nous obtenons les mêmes résultats qu'avec l'aspect généré manuellement, à savoir la détection de la fuite mémoire.

Extrait de code 5.3 : DSL requis pour créer automatiquement un aspect trouvant la fuite dans le programme jouet.

```
//// name : MyToyCounter
//// type : counter
//// className : B
```

Comme pour l'hypothèse précédente, nous passons ensuite à NOMAD, afin de nous assurer que notre approche marche aussi pour un programme réel. Pour cela nous retirons tout d'abord tous les aspects créés manuellement du dossier contenant le code source de NOMAD et inscrivons les deux lignes requises pour créer un aspect de type *counter* dans le fichier *Eval_Point.hpp*. Ce fichier contenant la définition de l'espace de nommage NOMAD dans lequel se trouve la classe concernée, nous n'avons pas besoin de le préciser dans la description de l'aspect. Nous exécutons ensuite ACRE sur le dossier contenant le code source de NOMAD et l'aspect présenté dans l'extrait de code 5.4 est généré dans le fichier *EVPCG.ah*, créé dans

le même dossier que les fichiers *.hpp* et *.cpp*. Nous pouvons y voir, aux lignes 10 à 13, que les lignes requises pour initialiser l'attribut de l'aspect sont bien indiquées au développeur.

Extrait de code 5.4 : Aspect généré automatiquement permettant la découverte d'une fuite mémoire dans NOMAD.

```
1  #ifndef __GENERATED_ASPECT_EVPCG_AH__
2  #define __GENERATED_ASPECT_EVPCG_AH__
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  aspect EVPCG {
8     public:
9        static int _Eval_PointCount;
10       // Do not forget to initialize this variable in the source code !!
11       // Just copy these lines into the appropriate .cpp file
12       // #include "EVPCG.ah"
13       // int EVPCG::_Eval_PointCount = 0;
14
15       pointcut Eval_PointCounted() = "NOMAD::Eval_Point";
16
17       advice Eval_PointCounted() : slice struct{
18          class Eval_PointCount{
19             public:
20                Eval_PointCount(){ EVPCG::_Eval_PointCount++;}
21             public:
22                ~Eval_PointCount(){ EVPCG::_Eval_PointCount--;}
23          } Eval_Point_counter;
24       };
25
26       advice execution("% main(...)") : after(){
27          printf("Final count of Eval_Point : %d\n", _Eval_PointCount);
28          if(_Eval_PointCount > 0)
29             printf("Memory leak !!\n");
30       }
31 };
32
33 #endif
```

Une fois cet aspect, et uniquement cet aspect, tissé avec le code source de NOMAD, nous obtenons le même résultat que celui affiché sur la figure 5.3. Cela nous permet de valider notre deuxième hypothèse : nous pouvons générer automatiquement des aspects depuis le code source d'un programme, en utilisant ACRE et le DSL que nous avons proposé, afin de tester de vrais programmes.

5.5 VALIDATION DE L'HYPOTHÈSE 3

Pour notre dernière hypothèse, formulée *les aspects ainsi générés peuvent être utilisés pour aider à la réalisation de tests d'interférence ?*, nous avons utilisé l'AOP pour obtenir les temps d'accès en lecture et en écriture de variables. Ces temps sont ensuite utilisés par l'approche Bhattacharya *et al.* (2012) pour aider à la détection de bogues de type interférence. Pour cela, nous nous sommes concentrés uniquement sur NOMAD. Dans un premier temps sur la version séquentielle, pour vérifier que nous pouvons effectivement obtenir les temps de lecture et d'écriture des variables souhaités. Dans un deuxième temps, nous nous sommes concentrés sur la version parallèle de NOMAD, qui utilise MPI (*Message Passing Interface*[2]). Pour ces deux étapes nous avons utilisé des aspects tout d'abord générés manuellement, puis automatiquement.

Une fois trouvé un attribut utilisé un très grand nombre de fois dans NOMAD, en l'occurrence l'attribut *mesh_index* de la classe *Mesh*, nous avons regardé au niveau des spécifications d'AspectC++ pour voir s'il est possible de déclarer une coupe transverse directement sur l'attribut. Cependant, même si ceci est possible, il est impossible par la suite de déclarer une méthode d'aspect sur la coupe transverse ainsi créée. En effet, dans la version actuelle d'AspectC++, les méthodes d'aspect sont faites pour être liées à des coupes transverses dynamiques (des méthodes, via les actions *call* et *execution*) ou pour des classes (pour leur ajouter des attributs, méthodes ou classes de bases). On ne peut définir une méthode d'aspect directement sur un attribut. Il nous faut passer par ses *getters* et *setters*. Ceci est donc une limitation de notre approche, mais qui vient d'AspectC++ et de l'utilisation explicite des pointeurs en C++. Une deuxième limitation de notre approche pour cette hypothèse vient elle aussi d'AspectC++. En effet, dans la version actuelle, nous ne pouvons accéder aux variables, locales ou globales, via des coupes transverses. Nous devons donc nous contenter d'avoir les accès en lecture et en écriture des attributs possédant un accesseur et une méthode de réglage.

Conscient de ces limitations, nous construisons donc un aspect capable de nous indiquer les temps souhaités, en déclarant deux méthodes d'aspect, une étant appelée juste après chaque exécution de l'accesseur souhaité, et l'autre en faisant de même pour la méthode de réglage. Ces méthodes d'aspects, visibles dans l'extrait de code 5.6, mesurent le temps auquel l'appel à la méthode a été effectué et l'affiche sur la sortie courante.

Nous tissons donc cet aspect avec la version séquentielle de NOMAD, exécutons cette dernière et obtenons, comme souhaité, les temps d'accès en lecture et en écriture de l'attribut

2. `http://www.mcs.anl.gov/research/projects/mpi/`

mesh_index. Nous générons ensuite cet aspect en utilisant notre DSL (extrait de code 5.5) et ACRE et obtenons l'aspect présenté dans l'extrait de code 5.6. Lorsque nous le tissons seul dans le code source de NOMAD, que ce soit pour la version séquentielle ou parallèle, nous obtenons le résultat attendu : l'AOP nous permet d'obtenir les temps d'accès, en lecture et en écriture, des attributs accessibles via un accesseur et une méthode de réglage. Ces temps sont ensuite fournis en entrée d'un autre approche. Celle ci nous donne la place optimale où insérer un délai dans NOMAD, ainsi que la durée de se délai. Une fois celui ci inséré, nous exécutons la version parallèle de NOMAD et nous voyons qu'effectivement un bogue de type interférence peut apparaître dans la version parallèle de NOMAD. Ceci nous permet donc de valider notre troisième hypothèse.

Extrait de code 5.5 : Lignes de code DSL requise pour générer l'aspect 5.6.

```
//// name : TimingAspect
//// type : timing
//// time : nanosecond
//// attribute : mesh_index
```

Extrait de code 5.6 : Aspect de type *timing* généré automatiquement.

```
#ifndef __GENERATED_ASPECT_TIMING_MESH_INDEX_AH__
#define __GENERATED_ASPECT_TIMING_MESH_INDEX_AH__

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace std;

aspect timingMeshIndex {

   public:
      struct timeval courant;
      long timeCourant;

      advice execution("% NOMAD::Mesh::set_mesh_index(...)"): after() {
         gettimeofday(&courant, NULL);
         timeCourant = courant.tv_usec;
         printf("Ecriture de mesh_index : %ld \n", timeCourant);
      }

      advice execution("% NOMAD::Mesh::get_mesh_index(...)") : after() {
         gettimeofday(&courant, NULL);
         timeCourant = courant.tv_usec;
         printf("Lecture de mesh_index : %ld \n", timeCourant);
      }
};

#endif
```

5.6 DISCUSSION DES RÉSULTATS

Pour montrer les résultats obtenus et valider les hypothèses nous avons, dans les sections précédentes, utilisé uniquement les aspects de type *counter* et *timing*, qui nous permettent de réaliser des tests de mémoire et d'interférence, car ils sont les plus généralisables. Avec seulement quelques modifications dans leur description (voir aucune dans le cas de certains compteurs) ces aspects peuvent être utilisés pour n'importe quel programme. Nous allons maintenant nous concentrer sur un autre type d'aspects que peut générer ACRE.

L'aspect de type *checking* présenté dans l'extrait de code 5.7, généré à partir des lignes DSL écrites dans l'extrait 4.4, permet de réaliser un test d'invariant : la vérification qu'une condition de l'algorithme de NOMAD est toujours satisfaite. Ce point concerne la taille du treillis de l'algorithme, essentiel pour la convergence de la méthode (voir Audet et Dennis, Jr. (2006)). Ce point, bien qu'essentiel, n'est jamais vérifié explicitement dans la version finale de NOMAD, pour ne pas rajouter des calculs supplémentaires, et est supposé valide et satisfait implicitement. L'utilisation de l'AOP pour tester ce point est donc utile : nous n'avons pas besoin de modifier le code source de NOMAD pour avoir une version test. Nous n'avons pas été en mesure de trouver un bogue avec l'aspect présenté dans l'extrait 5.7. Cela veut dire que, pour tous les problèmes que nous avons testé, ce point de l'algorithme est valide. Cependant, afin de nous assurer que si jamais un bogue est présent, un aspect de type *checking* nous permettrait de le trouver, nous avons muté le code source de NOMAD pour y introduire une mauvaise implémentation de l'algorithme et donc un bogue. Cette mutation s'est faite simplement en inversant un opérateur (l'opérateur + a été remplacé par l'opérateur −). Nous avons ré-exécuté NOMAD sur les mêmes probèmes que précédemment et, cette fois ci, l'aspect 5.7 nous a indiqué qu'il y avait des erreurs dans les résultats, comme attendu. Ceci montre qu'un aspect de type *checking*, généré automatiquement par ACRE, permet de trouver des bogues dans un programme C++.

Extrait de code 5.7 : Aspect de type *checking* généré automatiquement.

```
#ifndef __GENERATED_ASPECT_DELTAS_AH__
#define __GENERATED_ASPECT_DELTAS_AH__

#include <stdio.h>
#include <stdlib.h>
#include "Mads.hpp"

aspect Deltas{

   advice execution("% ...::iteration(...)") : after(){
      NOMAD::Signature * s = tjp->that()->_p.get_signature();
      int n = s->get_n();
      NOMAD::Mesh mesh = s->get_mesh();
      int l = mesh.get_mesh_index();
      NOMAD::Point delta_m(n), delta_p(n);
      mesh.get_delta_p(delta_p, l);
      mesh.get_delta_m(delta_m, l);

      for(int i = 0 ; i < n ; ++i){
         if(delta_m[i] > delta_p[i]){
            printf("Error : delta_m[%i] > delta_p[%i]\n", i, i);
            exit(0);
         }
      }
   }
};

#endif
```

Chapitre 6

CONCLUSION

Nous avons proposé une approche nouvelle, l'utilisation d'aspects générés automatiquement pour tester des programmes C++. L'utilisation de l'AOP pour les tests n'avait à ce jour été faite que pour les programmes écrits en Java ou pour des programmes embarqués écrits en C++ et la génération automatique des aspects n'avait été proposée que pour Java. Notre approche et les travaux réalisés lors de ma maîtrise sont synthétisés à la section 6.1. Les limitations et les menaces affectant notre approche sont résumées à la section 6.2. Finalement la section 6.3 présente les améliorations futures à apporter à notre approche.

6.1 SYNTHÈSE DES TRAVAUX

Afin de pouvoir générer automatiquement des aspects, nous avons développé un DSL, présenté sous forme de commentaires afin de ne pas modifier le comportement du code source. Ce DSL, qui permet de décrire l'aspect à générer, est la seule modification, à part dans le cas des aspects de type *counter*, à apporter au code source du programme afin d'avoir des aspects permettant de le tester. Nous avons aussi développé ACRE, un logiciel qui permet de lire les lignes de DSL et qui génère automatiquement l'aspect correspondant à la description. Les quatre types d'aspects supportés par notre approche permettent l'accès à différentes manières de tester un programme écrit en C++, telles les tests de mémoire, d'invariant et d'interférence.

Pour les aspects de type *counter*, nous avons montré qu'ils permettent de détecter des fuites mémoires, tant dans un programme jouet créé pour cela que dans un logiciel réel utilisé en recherche et en industrie, NOMAD. Les aspects de types *timing* permettent, quant à eux, d'obtenir les données nécessaires pour appliquer une approche permettant la détection de bogues de type interférence. Les aspects de type *logging* permettent d'obtenir une trace du programme et les aspects de *checking* autorisent une plus grande liberté au développeur. Tous ces aspects permettent au développeur de pouvoir tester son logiciel sans avoir à modifier le comportement du code source, les aspects étant créés dans des fichier séparés et les lignes de code DSL introduites dans le code source du programme ne modifiant pas le comportement de ce dernier.

Notre approche permet ainsi à des développeurs n'ayant aucune connaissance en programmation orientée aspect d'utiliser cette technologie afin de tester leur programme. Elle permet

aussi aux développeurs C++ de pouvoir tester leurs programmes sans avoir à modifier leurs code source, même s'ils n'ont pas de connaissances approfondies en techniques de test.

6.2 LIMITATIONS DE LA SOLUTION PROPOSÉE

Notre solution possède quelques limitations. Comme décrit dans la section 5.6, l'une d'entre elle (le fait de ne pouvoir, pour les aspects de type *timing*, avoir accès qu'aux attributs possédant un *getter* et un *setter*) vient d'une limitation propre au modèle d'exécution d'AspectC++.

Une autre limitation, qui représente une menace à la validité interne de notre approche, est le fait que les bogues détectés grâce aux aspects peuvent être causés par la présence des aspects dans le programme et ne pas être là à l'origine. En ce qui concerne les bogues que nous avons trouvés et exposés dans ce mémoire, nous avons montrés en section 5.6 que c'étaient bien des bogues présents dans NOMAD. Il convient donc aux utilisateurs de notre approche de vérifier que les aspects qu'ils créent n'introduisent pas de bogue dans leur programme. Ceci est essentiellement vrai pour les aspects de type *checking*, pour lesquels l'utilisateur a beaucoup de liberté au niveau de leurs contenus.

Une menace à la validité externe, et aussi une limitation, qui peut toucher notre approche est le fait que nous avons pour le moment testé cette dernière uniquement sur un programme jouet et sur NOMAD. Cependant, les aspects créés sont écrits en AspectC++ et peuvent donc être tissés dans le code source de n'importe quel programme écrit en C++. De plus, leur création repose sur un DSL qui n'affecte pas le code source dans lequel il se trouve. Enfin, comme mentionné dans la section 5.2, NOMAD est un logiciel fonctionnant sur les trois principaux systèmes d'exploitations (Windows, Mac OS et Linux) et qui est utilisé aussi bien en recherche que dans l'industrie. Nous pensons donc que notre approche peut être utilisée pour tester n'importe quel programme écrit en C++.

Une autre menace à la validité externe qui concerne notre approche est le fait que nous avons utilisé ACRE et les aspects générés uniquement sous Linux (et plus précisément Ubuntu). Cependant, ACRE est écrit en Java et peut donc être exécuté sur n'importe quel système d'exploitation possédant la machine virtuelle Java appropriée (en l'occurrence la version 1.6). Des tisseurs pour AspectC++ étant aussi disponibles pour tous les systèmes d'exploitations, nous pensons que notre approche peut elle aussi être utilisée sur tous les systèmes d'exploitation.

Finalement, concernant la fiabilité de notre approche, tous les programmes utilisés pour ce mémoire, que ce soit le programme jouet, ACRE ou NOMAD, ont leur code source disponible

sur internet. ACRE et le programme jouet sont sous licence GPL [1], et NOMAD sous licence LGPL [2].

6.3 AMÉLIORATIONS FUTURES

La première amélioration que nous souhaiterions apporter à notre approche est l'ajout de la gestion de nouveaux types d'aspects par ACRE et le DSL. Pour l'instant seuls quatre types sont disponibles (*counter*, *logging*, *checking* et *timing*) et ils ne permettent pas de faire tous les types de tests possibles.

Une autre amélioration possible est l'extension d'ACRE à la génération d'aspects écrits en AspectJ, afin de pouvoir utiliser notre approche pour des programmes écrits en Java. Nous devons cependant voir si nous conservons le DSL sous sa forme actuelle pour le code source Java ou non. Ceci amène à une autre amélioration possible, qui serait d'enlever les lignes de code DSL du code source et de les mettre dans des fichiers à part.

La dernière amélioration à apporter est l'ajout d'une interface graphique. Pour le moment ACRE s'utilise uniquement en ligne de commande et n'est donc pas très convivial. L'ajout d'une interface graphique pourrait rendre son utilisation plus simple. Une interface graphique simplifiant l'écriture du DSL pourrait aussi aider à l'utilisation de notre approche.

1. `http://web.soccerlab.polymtl.ca/~ducloset/ACRE`
2. `http://www.gerad.ca/nomad`

RÉFÉRENCES

AUDET, C. et DENNIS, JR., J. (2006). Mesh adaptive direct search algorithms for constrained optimization. SIAM Journal on Optimization, 17.

BEN-ASHER, Y., FARCHI, E. et EYTANI, Y. (2003). Heuristics for finding concurrent bugs. Proceedings of the 17th International Symposium on Parallel and Distributed Processing. 288.1–.

BHATTACHARYA, N., EL-MAHI, O., DUCLOS, E., BELTRAME, G., ANTONIOL, G., LE DIGABEL, S. et GUÉHÉNEUC, Y.-G. (2012). Optimizing threads schedule alignments to expose the interference bug pattern. Proceedings of the 4th edition of the Symposium on Search Based Software Engineering (SSBSE).

BOEHM, H.-J. (2012). Advantages and disadvantages of conservative garbage collection. Tiré de `http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html`, consulté le 06 juin 2012.

BRADLEY, J. (2003). An Examination of Aspect-Oriented Programming in Industry. Thèse de doctorat, Colorado State University.

D'AMORIM, M. et HAVELUND, K. (2005). Event-based runtime verification of java programs. Proceedings of the third international workshop on Dynamic analysis. WODA '05.

DAPER (2012). Linux programming blog : C++ object's reference counting, consulté en 2011. Tiré de `http://www.linuxprogrammingblog.com/cpp-objects-reference-counting`, consulté le 06 juin 2012.

DUCLOS, E., GUÉHÉNEUC, Y.-G. et LE DIGABEL, S. (2012). ACRE : An Automated Aspect Creator for Testing C++ Applications. Submitted in July 2012 to the IEEE Working Conference on Reverse Engineering (WCRE).

EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G. et UR, S. (2003). Framework for testing multi-threaded java programs. Concurrency and Computation : Practice and Experience.

ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S. et XIAO, C. (2006). The daikon system for dynamic detection of likely invariants. Science of Computer Programming.

FARHAT, S., SIMCO, G. et MITROPOULOS, F. J. (2010). Using aspects for testing non-functional requirements in object-oriented systems. Proceedings of the IEEE SoutheastCon 2010. 356–359.

GIBBS, T. H., MALLOY, B. A. et POWER, J. F. (2002). Automated validation of class invariants in c ++ applications. Proceedings of the 17th IEEE international conference on Automated software engineering. ASE '02, 205–.

GRADY, R. (1999). An economic release decision model : Insights into software project management. Proceedings of the Application of Software Measurement Conference.

HEINE, D. L. et LAM, M. S. (2003). A practical flow-sensitive and context-sensitive C and C++ memory leak detector. PLDI '03.

JOSHI, P., NAIK, M., PARK, C.-S. et SEN, K. (2009). Calfuzzer : An extensible active testing framework for concurrent programs. Proceedings of the 21st International Conference on Computer Aided Verification. 675–681.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., MARC LOINGTIER, J. et IRWIN, J. (1997). Aspect-oriented programming. ECOOP.

KNAUBER, P. et SCHNEIDER, J. (2004). Tracing variability from implementation to test using aspect-oriented programming. International Workshop on Software Product Line Testing (SPLiT 2004), Boston, Massachusetts, USA.

LE DIGABEL, S. (2011). Algorithm 909 : NOMAD : Nonlinear optimization with the MADS algorithm. ACM Transactions on Mathematical Software, 44.

LE DIGABEL, S. et TRIBES, C. (2011). Nomad user guide version 3.5.1. Gerad.

LI, X. et XIE, X. (2009). Research of software testing based on AOP. Proceedings of the 3rd International Conference on Intelligent Information Technology Application. 187–189.

LIENZ, B. P. et SWANSON, E. F. (1980). Software maintenance management. Addison Wesley.

METSÄ, J., KATARA, M. et MIKKONEN, T. (2007). Testing non-functional requirements with aspects : An industrial case study. Proceedings of the Seventh International Conference on Quality Software. 5–14.

METSÄ, J., KATARA, M. et MIKKONEN, T. (2008). Comparing aspects with conventional techniques for increasing testability. Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation. 387–395.

MUSUVATHI, M., QADEER, S. et BALL, T. (2007). Chess : A systematic testing tool for concurrent software.

NIELSON, F., NIELSON, H. R. et HANKIN, C. (1999). Principles of Program Analysis. Springer-Verlag New York, Inc.

PARK, S., LU, S. et ZHOU, Y. (2009). Ctrigger : exposing atomicity violation bugs from their hiding places. SIGPLAN Not.

PESONEN, J. (2006). Extending software integration testing using aspects in Symbian OS. Proceedings of the Testing : Academic & Industrial Conference on Practice and Research Techniques. 147–151.

PESONEN, J., KATARA, M. et MIKKONEN, T. (2005). Production-testing of embedded systems with aspects. Haifa Verification Conference. 90–102.

SCHILDT, H. (2004). The Art of C++. McGraw-Hill Osborne Media.

SIOUD, A. (2006). Gestion de cycle de vie des objets par aspects pour C++. Mémoire de maîtrise, Université de Québec à Chicoutimi.

SOFTWARE QUALITY RESEARCH GROUP, FACULTY OF SCIENCE, U. (2012). Concurrency anti-pattern catalog for java. Tiré de `http://faculty.uoit.ca/bradbury/concurr-catalog/`, consulté le 06 juin 2012.

SPINCZYK, O., GAL, A. et SCHRÖDER-PREIKSCHAT, W. (2002). Aspectc++ : An aspect-oriented extension to c++. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002).

SPINCZYK, O., LOHMANN, D. et URBAN, M. (2005). AspectC++ : Extension de la programmation orienté aspect pour C++. Software Developer's Journal.

STÖRZER, M., FORSTER, F. et STERR, R. (2006). Detecting precedence-related advice interference. 21st IEEE/ACM International Conference on Automated Software Engineering.

XIE, T. et ZHAO, J. (2006). A framework and tool supports for generating test inputs of aspectj programs. In Int. Conf. AspectOriented Software Development. 190–201.

XU, G., YANG, Z., HUANG, H., CHEN, Q., CHEN, L. et XU, F. (2004). Jaout : Automated generation of aspect-oriented unit test. Proceedings of the 11th Asia-Pacific Software Engineering Conference. 374–381.

XU, R.-G., GODEFROID, P. et MAJUMDAR, R. (2008). Testing for buffer overflows with length abstraction. ISSTA '08.

ZENG, F., CHEN, M., YIN, K. et WANG, X. (2009). Research on buffer overflow test based on invariant. Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on. 234 –238.

ZHOU, Y., ZIV, H. et RICHARDSON, D. J. (2004). Towards a practical approach to test aspect-oriented software. SOQUA/TECOS. vol. 58, 1–16.